8

# The CMU Design Automation System

## An Example of Automated Data Path Design

Def #5

A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, J. Kim

Carnegie-Mellon University
Departments of Electrical Engineering and Computer Science
Pittsburgh, Pennsylvania 15213

## Abstract

This paper illustrates the methodology of the CMU Design Automation System by presenting an automated design of the PDP-8/E data paths from a functional description. This automated design (using synthesis techniques) is compared both to DEC's implementation and the Intersil single chip implementation.

## 1. Introduction

As it is becoming possible to integrate larger numbers of logic components on a single chip, the need for more powerful design aids is becoming apparent. Indeed, these aids must be capable of supporting a designer from the system level of design down to the mask level. In this way the systems level designer can become more aware of the implications of higher-level design tradeoffs on implementation properties such as silicon area, power consumption, testability, and speed, and be able to make more timely use of new technologies. The ultimate goal of the Carnegie-Mellon University Design Automation (CMU-DA) System [12] is to provide a technology-relative, structured-design aid to help the hardware designer explore a larger number of possible design implementations. Inputs to the system are a behavioral description of the system to be designed, an objective function which specifies the user's optimization criteria, and a data base specifying the hardware components available to the design system.

The CMU-DA system differs from other design automation systems because the input design description is a functional specification. Such a specification provides a model that, while accurately characterizing the input-output behavior desired for the implementation, does not necessarily specify its internal structure. The system software collectively performs the synthesis function by transforming the input functional description into a structural description. The design process involves binding implementation decisions in a top-down manner as a design proceeds through the design system. More structural decisions are made at each level until a complete hardware specification is obtained, with the most influential design trade offs being performed first in order to cut down the design search space.

The purpose of this paper is to illustrate the methodology of the CMU-DA system. The results given here are worst case - many optimizations which are straightforward have not been implemented yet; research is in progress on others. The design of the data part of a DEC PDP-8/E [5] from the ISP level through to a TTL and standard cell design will be discussed. Only the subset of the full DA system which is presently implemented has been used for this example. The components are shown in Figure 1.

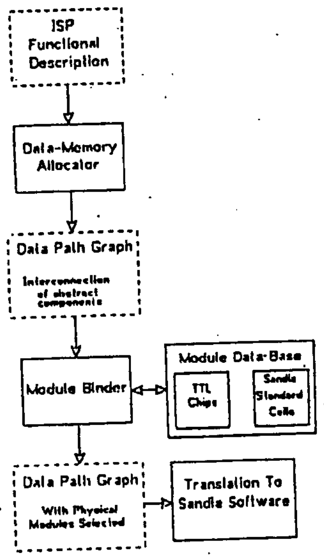

ISP Functional Description
Data-Memory Allocator
Data Path Graph
Interconnection of abstract components
Module Binder
Module Data-Base
TTL Chips
Sandia Standard Cells
Data Path Graph
With Physical Modules Selected
Translation To Sandia Software

Figure 1: The CMU Design System

The PDP-8/E is first functionally described using the ISP language [1]. A data-memory allocator [7] is used to generate the structure of the data paths from the functional description. This allocation is in terms of abstract logic components. The next step in the design is to bind physical modules to the abstract design from a module database [9]. This step provides the system with the capability of designing relative to new technologies. This binding will be illustrated both in terms of TTL chips and the CMOS standard cells in the Sandia design system [10]. The standard cell output of the CMU-DA system is then translated for input to the Sandia design system. At this point, chip area can be calculated and detailed timing information can be gathered using SALOGS [3].

The paper will conclude by comparing these two alternative designs to commercial implementations.

This research is supported in part by NSF Grant MCS77-09730 and Army Research Office Grants DAAG29-76-G-0224 and DAAG29-78-G-0070.

## 2. The PDP-8/E Example

As the CMU-DA system has evolved, more complex design examples have been used to observe its performance. The use of the PDP-8/E is a major step in two ways: 1) it represents about a five fold increase in the circuit complexity over previously reported example designs [8], and 2) it is a commercially available system that has been implemented in various technologies by different manufacturers over several years. Thus it is possible to compare a non-trivial automated design to designs done by several designers with different logic components.

The PDP-8/E is one model in a family of computers having nearly identical ISP descriptions. (That is, they implement nearly the same instruction set, with different hardware structures). A portion of the ISP description used by the automated design system is shown in Figure 2.

The description begins by declaring the memory and processor state. A 13 bit link-accumulator (lac) register is defined but can alternately be accessed as the one bit link (L) and 12 bit accumulator (Ac). Next, instruction interpretation is defined using the declared memory and registers. After the instruction fetch and increment of the program counter in the instruction interpretation section, instruction execution (exec()) is called. Illustrated here are the six memory reference instructions. Not shown but implemented in the automated designs are the effective address calculation, input/output instructions, and the operate microinstructions. Note that the Mb=Mp[Pc] instruction fetches the location pointed to by the Pc and places it in the Mb register. No mention need be made of transfer of the current Pc to a memory address register, which is a part of the hardware structure.

The ISP description [1] is compiled into a representation which is machine readable by the data-memory allocator. The next sections will discuss the data-memory allocation (where an abstract data path is synthesized), a module binder that illustrates how the CMU-DA system can design relative to new technologies, and a translator to Sandia's SALOGS simulator [10].

## 3. The Allocation Process

The data-memory allocators perform a mapping function from the algorithmic (ISP) description to the data-path part of the hardware implementation, which is called a data-path graph. The data part consists of the data-storage elements, data operators, and data paths necessary to implement the operations specified in the algorithmic description. Due to the characteristics of the ISP language this mapping may be multi-valued in either direction, rather than a simple one-to-one translation.

The PDP-8/E design described here was produced by a data-memory allocator which uses the distributed-logic design style. This style of (or approach to) design encompasses design with small and medium scale integration components. As pointed out earlier, the allocator itself is technology relative and the mapping onto specific integrated circuit packages is performed by a separate module binder program. The process referred to as allocation throughout the remainder of this paper is a synthesis of logic using generic logic elements: data paths, operators, registers, and multiplexers, all of any bit width.

The procedure used by the allocator might be compared to a two-pass compilation. The first pass may be considered a syntax or feasibility check. The allocator inputs a parsed ISP description, constructs data structures analogous in function to symbol tables, and enforces constraints necessary to insure that the data-storage locations, logical mappings, and

```
pdp8 :=
    BEGIN

! The basic PDP-8 instruction set (without extended arithmetic
! element) is implemented.  No I/O devices are included in the
! description.

! MP State

    Mp[#0:#7777]<0:11>,                          ! Main memory
    Mb<0:11>,                                    ! Memory buffer

! PC State

    lac<0:12>,                                   ! Link  ac register
        L<>         := lac<0>,                   ! Link bit
        Ac<0:11>    := lac<1:12>,                ! accumulator
    Pc<0:11>,                                    ! Program counter
    last.pc<0:11>,

! Instruction Interpretation

    start :=
        BEGIN
        go = 1 NEXT
        run()
        END,

    run\instruction.interpretation :=
        BEGIN
        IF go =>
            BEGIN
            Mb = Mp[Pc]; last.pc = Pc NEXT
            Pc = Pc + 1 NEXT
            exec() NEXT
            IF interrupt.enable AND interrupt.request =>
                BEGIN
                Mp[0] = Pc NEXT
                Pc = 1
                END NEXT
            RESTART run
            END
        END,

! Instruction Execution

    exec\instruction.execution :=
        BEGIN
        Ir = Mb<0:2> NEXT
        IF (Ir GEQ #3) AND (Ir LEQ #5) => ma() NEXT
        IF Ir LLQ #2 => Mb = Mp[ma()] NEXT
        DECODE Ir =>
            BEGIN
            #0 := and.:= Ac = Ac AND Mb,             ! And
            #1 := tad := lac = lac + Mb,             ! ADD (TC)
            #2 := isz := BEGIN                       ! Increment and
                                                     ! skip if zero
                         Mb = Mb + 1 NEXT
                         IF Mb EQL 0 => Pc = Pc + 1
                         END,
            #3 := dca := BEGIN                       ! Deposit and
                         Mb = Ac NEXT                ! clear Ac
                         Ac = 0
                         END,
            #4 := jms := BEGIN                       ! Jump to
                         Mb = Pc NEXT                ! subroutine
                         Pc = ma + 1
                         END,
            #5 := jmp := Pc = ma,                    ! Jump
            #6 := iot(),                             ! I/O execution
            #7 := opr()                              ! Operate
                                                     ! microinst.
            END NEXT
        IF (Ir GEQ #2) AND (Ir LEQ #4) => Mp[ma] = Mb
        END
    END         ! End of description
```

Figure 2: ISP Description of the PDP-8/E

input/output interface characteristics specified in the description can be implemented in hardware. If no errors are encountered, it proceeds to allocate the basic data-storage structures called for in the description, and any additional data paths, storage, and operators necessary to implement variable-accessing schemes described by ISP. The second pass may be considered as the semantic phase, with the activity of code generation replaced by the allocation of data paths, operators, and additional storage as needed to implement the actions described in the ISP description. Parallelism analysis is performed at several levels to warn the user of error conditions and determine constraints relating to optimization of hardware. The allocation is then completed by the addition of multiplexing where required.

However, allocation differs from compilation in that in a compilation one is concerned with implementing the specified data operations on a fixed data part whose capabilities are known *a priori*. In allocation, the allocator must be able to recall and utilize the capabilities of a data part which is being dynamically created. The allocator thus works from the inside out, first creating the data storage and access structures, and then adding the necessary data paths and operators to perform the described data operations. Finally, the output of the allocator is a non-planar directed graph, rather than a linear list of compiled instructions.

The first version of the allocator is experimental, and it performs only minor optimizations on the allocated hardware. It has been designed to investigate the feasibility of performing the mapping from ISP to hardware, the types of data structures needed for allocation, and areas where optimizations are possible in future, more sophisticated allocators.

The allocator has been designed as a possible skeletal structure for future allocators in order to standardize input/output formats and data structures.

## 4. Performance of the Data-Memory Allocator

The allocator program was run using the ISP description of the PDP-8/E and the resultant data paths are shown in Figure 3. A binding of modules was done by hand to compare the results of the allocator to the original DEC design (Figure 4) [5].

It is difficult to compare the automated PDP-8/E data-path design with the original DEC design for three reasons. First, the ISP description input to the allocator declares as registers some values the PDP-8/E uses but never stores explicitly in registers, such as the effective address. These show up as registers in the allocator's design. Second, the allocator designs distributed logic, and the DEC design was done in the central-accumulator design style. (That is, this allocator does not contain the design rules for large scale collapsing of the data paths into a central-accumulator style of design [13]). Third, the DEC design has assumed a boundary between the control and data-memory parts of the design, but the boundary is different from that imposed on the allocator by the ISP description. Thus some tests, flags, and registers which must be declared explicitly in the ISP description are part of the control in the DEC design.

The main reason for the difference in design seen from the block diagram level of Figures 3 and 4 is that the design styles are different. The multiplexing is used in different ways. In the DEC version, the operators are shared, and are used to provide no-op paths from one register to another. In the CMU version, only registers are shared and use multiplexed inputs. The ISP language is partially the source of this disparity. In ISP, the user can repeatedly use register A as a destination from various sources. However, the expressions A+B and C+D do not imply nor discount a single adder. Other differences in the design include the use of multiplexers for shifting in the DEC design, and use of true/complement 0/1 chips for creating complements. "ORing" of the MQ and AC registers in the DEC version is done within the multiplexing hardware. Constants are often created in one place and gated over already existent data paths to the registers. In the CMU version, these constants are multiplexed at the register inputs.

In spite of these differences, estimates of chip count indicate that the allocator produces a path graph which would require 39% more integrated circuit chips that the DEC designers used for the data paths and registers. These estimates were done by hand to gauge the performance of the allocator; an automatic binding is discussed in the next section. These estimates were made using the same 1970 technology chip set the DEC designers had to deal with. The 39% excess hardware can be found in multiplexors which connect the registers, the extra registers declared in the ISP description, and duplicated operators like increment and add. Much of this excess can be attributed to the lack of optimization capability in the allocator algorithm. Future, more sophisticated allocation algorithms, coupled with the capability for high level optimization [12] available in the complete CMU-DA system will be able to significantly improve the data part design.

Further analysis of this design is in progress and includes a manual implementation of the control part. Comparisons of the DEC and CMU data-path speeds will then be possible.

## 5. Module Binding

The module binding phase of the CMU-DA system employs the Module Data Base System (MDBS) and follows the data-memory allocation step. It has the task of translating the abstract links, memories, registers, and operations in a data-path graph into a design using physically realizable modules. A second pass of module binding will occur after control allocation.

At present the module binding portion of the design system is primarily a research tool that will be used to investigate automated module binding. A goal of this research is to model the module binding problem sufficiently to generalize this part of the CMU-DA system to handle a wide range of module types from LSI chips through Standard Cells. The implemented portions of MDBS were used to assist a designer in binding data-part modules in the CMU PDP-8/E data-paths produced by the Data-Memory Allocator described in Section 4. The results of the data-part module binding are compared to the DEC PDP-8/E design in Section 7. A similar comparison will be made to the standard-cell binding after those results are presented in Section 8.

## 6. Organization of MDBS

MDBS consists of four sections shown in Figure 5: an I/O section that is responsible for translating between the internal and external forms of the path graph; a Module Data Base access mechanism; a command language interface; and the module binding mechanism.

The input/output section is the interface to other parts of the CMU-DA system. The path graph, generated by an allocator is placed in internal form for processing. The output file has the same format as the input path graph (with module binding information appended) and can be reread by the input section for additional processing.

The Module Data Base is a hierarchical data base that is distributed in various ASCII files. The highest level of the data base is the index which is read automatically during system initialization. The index contains pointers to all defined design-style sets, each of which is a collection of module sets appropriate for a given design style. A design style set file contains pointers to the actual module set information (the "Data Book") and summary information (typical cost, speed,

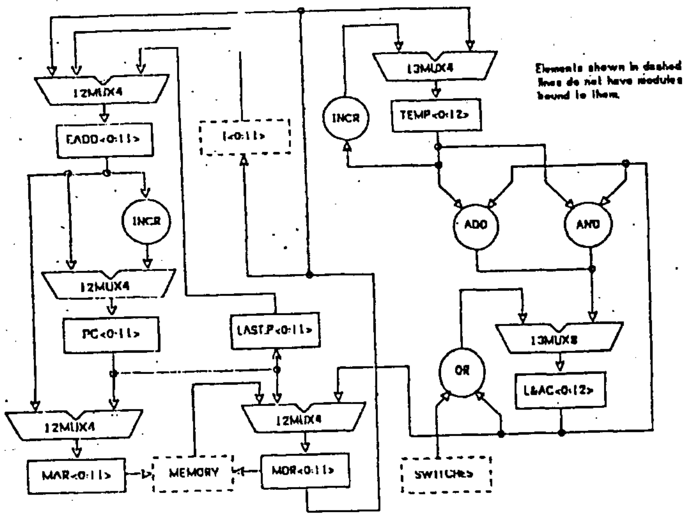


Figure 3: Allocator Generated PDP-8/E Data Paths

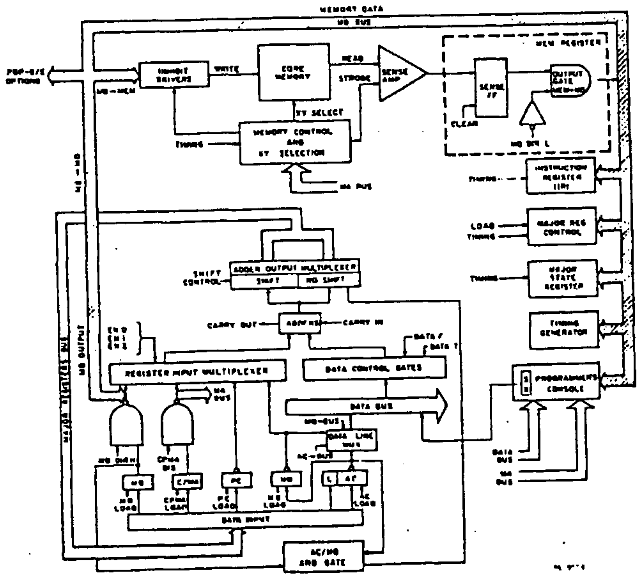


Figure 4: DEC PDP-8/E Data Paths

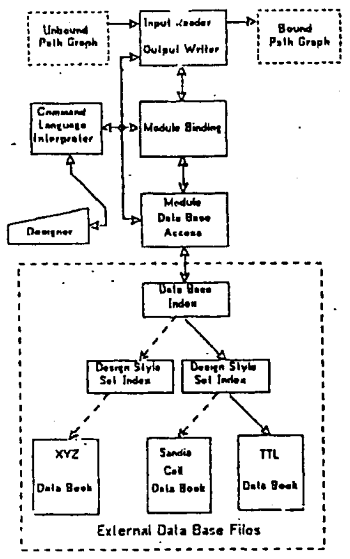


Figure 5: Organization of the Module Data Base System

load, and drive capabilities of modules in the set). Access to data base information during module binding occurs frequently and utilizes various levels of detail from basic type matching through specific delay and timing characteristics.

The Command Language Interpreter (CLI) is the experimenter/designer's interface to the module binding system. The CLI may be used only to select a file to be processed and direct the disposition of the resulting bound path-graph. The CLI also provides a number of tools to inspect the path-graph, modify the graph structure and bind designer selected modules to specific nodes of the graph.

The module binder applies transformations to both the path graph and module functions in order to match the desired behavior with the available building blocks. The graph transformations are localized decompositions or combinations of nodes that preserve the specific behavior. Module transformations are primarily combinations of nodes since modules cannot be physically decomposed. However, multifunction modules such as shift registers and ALUs may require partitioning of the non-conflicting functions of the same module onto separate nodes of the graph.

The most prevalent type of graph transform is localized to a single node or several connected nodes of a similar type. Registers are usually decomposed into nodes of smaller bit width. Logical operators are usually modified by reduction to a canonical form, then synthesized with available module operations. Multiplexors and demultiplexors are frequently transformed from a single level to a multilevel form. Arithmetic operators (particularly the signed and complement arithmetic modes) require algorithmic decomposition.

A more complex type of transformation involves the combination of nodes of different types into single nodes capable of multiple functions. Shift operators and special purpose arithmetic operators (increment, decrement, and clear) are generally combined with register functions in available module sets. These transformations often provide significant reductions in the graph complexity by elimination of constants and reduction of multiplexor size.

Operator node transformations primarily involve the application of axioms and identities to combine available modules into an aggregate that performs a desired function. Boolean identities and DeMorgan's Theorem will direct logical synthesis. Arithmetic mode transformations (for unsigned, signed-magnitude, two's complement, and one's complement) will be utilized to synthesize required modes from available modes.

For cases where single transformations on either the graph or the modules do not provide the desired match, an iterative approach will be utilized. The graph and the module transforms will be alternately applied until one or more matches are found, a cycle is detected in the transformations, no further gain is detected by applying transforms, or one of the system constraints (speed, cost, etc.) is violated by the resulting implementation.

A central goal of the CMU-DA system is to produce designs that have been optimized toward the designer's objectives and fall within the external constraints. Module binding is the first operation in the design system that attaches actual costs to the implementation and has specific speed, delay, and power information available. Therefore, evaluation of the bound design must be performed to insure compliance with the constraints. Critical constraints (i.e. constraints which the designs *must* meet) will be dynamically estimated by projecting the final value based on an extrapolation from the number of nodes remaining to be bound and the accumulated value for the nodes already bound; a true evaluation of the fully bound design will be made as a final pass to insure compliance with the constraints. The dynamic evaluation will be used to select between functionally identical module choices with different performance parameters.

## 7. PDP8 Baseline Example

A partial module binding of the PDP-8/E was done with the available pieces of MDBS, which contain limited transform capabilities.

The module binding section contains a set of primitive operations that can modify the path graph structure (but not its behavior). The operations allow register nodes to be split at bit boundaries, operator nodes to be joined into single nodes with wider bit widths, nodes to be inserted, and nodes to be deleted. These operations allow the path graph nodes to be transformed to conform with the structure of available modules. As the MDBS becomes more fully implemented, these primitive operations will be used to build larger scale path graph transforms that can be applied automatically.

The existing system aids the designer in producing correctly bound path graphs by the strict enforcement of rules concerning application of the structural modification primitives. These operations are restricted to minimize the possibility of modifying the behavior of the path graph. Some examples of these rules are:

- Nodes may not be deleted while connected to more than one link.
- Links may not be deleted while joining two nodes.
- Half links (i.e., signals to external sources) may not be deleted.
- Operator nodes may be joined only if they are of the same type.

A different but equally important kind of assistance is provided to allow the designer to display any aspect of the binding process. For example, the designer may display one module in the data base, all modules providing a specific function, or the entire data book.

The module choices were made by the designer using information from the path graph and a small TTL module data base from the distributed Design Style Set. The purpose of binding the data part with the existing system is to contrast the module selection with a hand designed PDP-8/E. This example provides a worst-case from which to judge the performance of MODS as more capabilities are added.

Figure 6 was generated by the MODS. The registers (named variables), operators, and multiplexors are identified in the "Comp." (Component) column. The "Device" column lists the name of the selected package. "Mods." lists the number of modules required to implement the component function. The term "module" refers to a separate functional unit in this context. There may be several modules contained in one package. "Pkgs." lists the number of packages required for each function. "Gates" is an estimate of the equivalent number of logic gates to implement the function. The percentage of the total gates required is listed in the "% Total Gates" column. The cost of each function implementation is computed as the basic cost for the number of packages required plus an overhead mounting cost of $3.00 per package. The percentage of the total cost attributed to each function is listed in the "% Total Cost" column.

| Comp. | Device | Mods. | Pkgs. | Gates | %Total Gates | Cost | %Total Cost |
|---|---|---|---|---|---|---|---|
| ENDG | SN74174 | 2 | 2 | 76 | 3.98 | 7.78 | 2.58 |
| LAC | SN74194 | 3 | 3 | 105 | 5.50 | 11.67 | 3.87 |
| LAST.P | SN74174 | 2 | 2 | 76 | 3.98 | 7.78 | 2.58 |
| PC | SN74161 | 3 | 3 | 204 | 10.69 | 11.67 | 3.87 |
| MDR | SN74174 | 2 | 2 | 76 | 3.98 | 7.78 | 2.58 |
| MAR | SN74174 | 2 | 2 | 76 | 3.98 | 7.78 | 2.58 |
| TEMP<8> | SN7474 | 1 | 1 | 6 | 0.31 | 3.35 | 1.11 |
| TEMP | SN74174 | 2 | 2 | 76 | 3.98 | 7.78 | 2.58 |
| EQL | SN7485 | 3 | 3 | 93 | 4.87 | 11.37 | 3.77 |
| EQL | SN7485 | 3 | 3 | 93 | 4.87 | 11.37 | 3.77 |
| INCR | SN7483 | 3 | 3 | 108 | 5.66 | 10.77 | 3.57 |
| INCR | SN7483 | 3 | 3 | 108 | 5.66 | 10.77 | 3.57 |
| AND | SN7408 | 12 | 3 | 12 | 0.63 | 9.60 | 3.18 |
| OR | SN7432 | 12 | 3 | 12 | 0.63 | 9.75 | 3.23 |
| ADD | SN7483 | 4 | 4 | 144 | 7.54 | 14.30 | 4.76 |
| 13MUX8 | SN74151 | 13 | 13 | 156 | 8.17 | 46.67 | 15.48 |
| 12MUX4 | SN74153 | 12 | 6 | 96 | 5.03 | 21.54 | 7.14 |
| 12MUX4 | SN74153 | 12 | 6 | 96 | 5.03 | 21.54 | 7.14 |
| 12MUX4 | SN74153 | 12 | 6 | 96 | 5.03 | 21.54 | 7.14 |
| 12MUX4 | SN74153 | 12 | 6 | 96 | 5.03 | 21.54 | 7.14 |
| 13MUX4 | SN74153 | 13 | 7 | 104 | 5.45 | 25.13 | 8.33 |
| Totals | | 131 | 83 | 1909 | 100.00 | 301.54 | 100.00 |

| Component Class | % Gates | % Cost |
|---|---|---|
| Registers | 36.40 | 19.17 |
| Operators | 29.86 | 23.49 |
| Multiplexers | 33.74 | 52.37 |

Figure 6: Module Utilization For PDP-8/E Data Part

The choice of registers differed from the hand module binding in a few locations. The hand bound PDP-8/E used SN74194s (four bit universal shift registers) exclusively while the MODS chose SN74174s (six bit D registers) when there was no requirement for the added shift capability. The Program Counter (PC) register was selected as three SN74161s (universal counters) based on the INC flag associated with the path graph node. The Accumulator (LAC) was first split into a one bit node and a twelve bit node, then the twelve bit part was allocated as three SN74194s. This is the same allocation that was done by hand. However, the choice only satisfies the shift ration flags (LSHFT and RSHFT). The increment requiremen (INC) has effectively been partitioned out and must be bound separately.

The package count was 30% higher for the MODS selection than for the DEC implementation (64 packages for DEC vs. 83 packages for MODS). This agrees closely with the results obtained by selecting modules strictly by hand (refer to Section 4). The 30% difference is attributable to the different design styles used and the allocator's implementation of the design. Comparing the total cost of modules for the DEC implementation and the MODS implementation (Figure 6), it is found that the costs also are 30% higher for the automated implementation, while the number of equivalent gates is 65% higher for the automated implementation. This indicates that MODS chose modules with a higher level of integration than DEC did.

A comparison of the percentage of equivalent gates and the percentage of cost accumulated in three functional classes (registers, operators, and multiplexors) indicates surprisingly uniform comparisons. The percentage of both gates and cost is higher for registers in the MODS implementation than in the DEC implementation. This trend is expected since the DEC PDP-8/E uses a central accumulator design style. Also, the slightly lower percentages for gates and costs in the operator class is reasonable for the DEC implementation. The most surprising comparison is the near identical percentage of gates devoted to data path routing (i.e. multiplexors) in the two designs implemented in different design styles. It would be expected that the central accumulator style would utilize more data path routing than the distributed style. This apparent anomaly is a clue to the area where the module binding can make local improvements in a path graph for distributed designs. By utilizing functions intrinsic to certain modules (such as the CLEAR on registers), constants and their associated data paths can be eliminated and improve the cost of implementing a design.

The TTL module binding using MODS compares favorably with the DEC implementation and previous hand module bindings of the automated path graph. It is expected that much improvement in the package count (and the cost) is forthcoming as transforms and evaluation techniques are implemented in MODS. However, TTL module binding is just one objective of a generalized design system. The following section discusses an approach to binding CMOS standard cells to a design with the objective of being able to automate and produce LSI designs.

## 8. Standard Cell Generation

The Sandia standard cell library [11] can also be used as physical modules to implement the automated PDP-8/E data part design. Then, using the Sandia software package [4, 10], it is possible to produce a simulation, insert faults, and perform automated cell placement and IC mask generation for a CMOS LSI chip implementation.

The standard cell binder used for this experiment was a small, automatic package which accessed a local data base of Sandia cells. This package only performed the essential transformations on the graph–expansion of nodes to match the standard cells. However, this package also gives us a worst-case measure for module binding performance.

The translation of the design from one environment, the module binder output, to another, the simulator input, involves both the expansion of multi-bit paths produced by the module binder to the single bit connection format of the simulator input and also the explicit identification of fan-out points. In addition to this latter process, termed resolving, the translator must generate gate specific parameters, such as propagation delays, by computing capacitances to obtain a realistic simulation using SALOGS. Delay parameters are inserted in the simulation model by the use of delay gates with associated times.

The input to SALOGS is a description of the network, written in NDL [3]. NDL describes the interconnection of functional blocks. Input and control signals are generated through the SALOGS simulation language SALSIM [4]. In SALOGS, gate representation includes built-in simulator elements such as inverters, transmission gates, NAND, AND, OR, and NOR gates. In addition, any set of elements can be defined as a functional block and the block used as a new element.

The NDL can then be used to automatically generate an IC mask and to determine chip area. The portion of the total data-path area taken up by the different modules is summarized in Figure 7.

| Component | Area | % of Total Area | Number of *Gates | % of Total Gates | % of Gate subtotal |
|---|---|---|---|---|---|
| PC | 5126.3 | 6.9 | 170 | 9.4 | 13.0 |
| LAC | 4377.1 | 5.9 | 117 | 6.5 | 8.9 |
| ADD | 3839.9 | 5.2 | 117 | 6.5 | 8.9 |
| OR | 432.0 | 0.6 | 18.0 | 1.0 | 1.4 |
| AND | 476.3 | 0.6 | 18.0 | 1.0 | 1.4 |
| IRLG3 | 1433.3 | 1.9 | 45 | 2.7 | 3.6 |
| INCR | 3544.6 | 4.8 | 108 | 8.0 | 8.3 |
| **13MUX4 | 4217.4 | 5.7 | 60 | 3.3 | 4.6 |
| 12MUX4 | 3991.6 | 5.3 | 56 | 3.1 | 4.3 |
| 12MUX4 | 3991.8 | 5.3 | 56 | 3.1 | 4.3 |
| 12MUX4 | 3991.4 | 5.3 | 56 | 3.1 | 4.3 |
| 12MUX4 | 3991.6 | 5.3 | 56 | 3.1 | 4.3 |
| 13MUX8 | 9183.4 | 12.4 | 139 | 7.7 | 10.6 |
| COND | 1321.0 | 1.8 | 45 | 2.5 | 3.4 |
| INCR | 3544.6 | 4.8 | 108 | 6.0 | 8.3 |
| LAST.P | 1323.0 | 1.8 | 45 | 2.5 | 3.4 |
| NOR | 1323.0 | 1.8 | 45 | 2.5 | 3.4 |
| NOR | 1323.0 | 1.8 | 45 | 2.5 | 3.4 |
| subtotal | 57067.5 | 77.2% | 1308 | 72.5% | 100.0% |
| A-AND2 | 317.5 | 0.4 | 12 | 0.7 | |
| ***3xEQL9 | 2501.0 | 3.3 | 74 | 4.1 | |
| 2x12MUX2 | 2644.2 | 3.6 | 50 | 2.8 | |
| 1x1MUX2 | 141.0 | 0.2 | 3 | 0.2 | |
| 2xEQL2 | 158.4 | 0.2 | 5 | 0.3 | |
| SWITCH | 1323.0 | 1.8 | 45 | 2.5 | |
| T | 1321.0 | 1.8 | 45 | 2.5 | |
| 2xEQL12 | 2210.6 | 3.0 | 64 | 3.5 | |
| LSS | 2460.9 | 3.3 | 84 | 4.7 | |
| GEQ | 2460.9 | 3.3 | 84 | 4.7 | |
| IRQ | 1103.8 | 1.5 | 32 | 1.7 | |
| 3xFLAG | 339.8 | 0.5 | 11 | 0.6 | |
| TOTAL | 74042.0 | 100.1% | 1817 | 100.8% | |

* Gate count is in terms of 2 input NAND gates
** 13MUX4 is 1 of 4 MUX with bit width of 13
*** nxEQL9 indicates n copies of 9 input EQL comparator

Figure 7: Module Data from Translator

The upper portion of the Figure compares the size of the data-path elements listed in Figure 3. The lower part of the table describes some modules that are more accurately defined as control than data-path. As expected, the percent of sub-total gate count in the upper portion of the table closely resembles the results from the TTL binding shown in Figure 6.

In sum, the CMU PDP-8 design required 74,042 mil-sqr, ignoring the area taken up by routing. The experience with Sandia's IC mask design system indicates that routing takes up about an additional 75% of the area occupied by the standard cells, yielding a chip area of 129,574 mil-sqr. By way of comparison, Intersil's one chip CMOS CPU implementation of the PDP-8 takes up 29,014 mil sqr [6]. It is estimated that 35% of Intersil's CPU chip is devoted to the functional elements equivalent to that generated by the CMU-DA system. Thus, there is a factor of 13 difference in the area required for the two designs.

A model can be devised to attribute this seemingly large difference to various parts of the design system. Since each part of the design system builds on top of the previous stage, a multiplicative model is used. This model must take into account the non-optimality of the allocator, non-optimality of the module binder, differences in basic feature size, and the differences in routing techniques. The result of the allocator section indicates a design requiring 1.3 times the size of the DEC design. There is also a difference in the basic feature size of the Intersil and Sandia technologies. By way of comparison, a 12 bit register implemented with Sandia's standard cells occupies 4 times the area of equivalent register in Intersil's design [2]. The multiplicative model then becomes

$$13.8 = (1.3)(4)R,$$

where R is a factor indicating a difference in size between the CMU design and the Intersil design introduced by the module binder. In this case R = 2.5. Not included in the model are factors due to difference between hand packed and channel routing techniques, nor factors considering that large structures (e.g. wide multiplexors) can be more optimally designed by hand than by combination of simple standard cells.

In the worst case, assuming similar input structures and feature size, the CMU module binder would produce a design taking 2.5 times the area of the Intersil design. However, as discussed above, there are other factors which increase the CMU design size that were not accounted for in the model.

## 9. Summary and Conclusions

The paper has illustrated the methodology behind the CMU Design Automation System. In particular, the datapath of a non-trivial digital system (PDP-8/E) has been designed from an ISPL functional description. Two types of physical modules were bound to the datapath design.

The binding using TTL series modules indicated that the CMU design required 30% more modules than the DEC implementation. The binding using CMOS standard cells indicated that the CMU design is at most a factor of 2.5 off, and due to differences in routing techniques may be actually closer in area to the Intersil design.

As a whole the system has demonstrated the synthesis function in digital system design. The allocator research indicates automated logic synthesis with optimization is feasible and specific module-set information is not necessary in order to produce a reasonable design. The module binding section has demonstrated how the system can design relative to new technologies. Future work with the design system will deal with optimization techniques to be used in better directing the design algorithms for more complex designs.

### References

1. Barbacci, M., Barnes, G., Cattell, R., Siewiorek, D. The Symbolic Manipulation of Computer Descriptions ; The ISPS Computer Description Language. Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., March, 1978.

2. Bell, G., C. Mudge, J.E. McNamara. *Computer Engineering*. Digital Press, 1978.

3. Case, G.R. and J.D. Stauffer. SALOGS-IV, Program to Perform Logic Simulation and Fault Diagnosis. Proceedings 15th Design Automation Conference, IEEE, 1978.

4. Case, G.R. and J.D. Stauffer. SALSIM - A Langu for Control of Digital Logic Simulation. Proceedings of the 11th Annual Asilomar Conference on Circuits, Systems, and Computers, IEEE Circuits and Systems Soc., IEEE Control Systems Soc., Naval Postgraduate School, Univ. of Santa Clara, November, 1977, pp. 370-373.

5. DEC Staff. *PDP-8/E Maintenance Manual.* Digital Equipment Corporation, 1972. DEC-8E-HR1B-D

6. Electronics Magazine Staff. CMOS, Moving Along. *Electronics 48*, 10 (May 1975), .

7. Hafer, L. Data-Memory Allocation in the Distributed Logic Design Style. Master Th., Carnegie-Mellon University, December 1977.

8. Hafer, L.J. and A.C. Parker. Register-Transfer Level Automatic Digital Design: *The Allocation Process.* Proceedings of the 15th Design Automation Conference, IEEE, 1978.

9. Leive, G.W. The Binding of Modules to Abstract Digital Hardware Descriptions. PhD Thesis Proposal, Carnegie-Mellon University, 1977.

10. Preas, B.T. and C.W. Gwyn. itecture For Contemporary Computer Aids to G erate IC Mask Layouts. Proceedings of the 11th Annual Asilomar Conference on Circuits, Systems, and Computers, IEEE Circuits and Systems Soc., IEEE Control Systems Soc., Naval Postgraduate School, Univ. of Santa Clara, November, 1977, pp. 309-317.

11. Sandia Staff. *Standard Cell User's Guide.* Sandia Laboratories, 1978.

12. Snow, E.A., D.P. Siewiorek and D.E. Thomas. A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations and Design Tradeoffs. Proceedings of the 15th Design Automation Conference, IEEE, June, 1978.

13. Thomas, D.E. and D.P. Siewiorek. Measuring Designer Performance to Verify Design Automated Systems. Proceedings of the 14th Design Automation Conference, IEEE, 1977, pp. 411-418

DEF012052

9

IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, VOL. CAD-6, NO. 2, MARCH 1987 Ref #2

# Flamel: A High-Level Hardware Compiler

HOWARD TRICKEY, MEMBER, IEEE

*Abstract*—This paper describes the design and implementation of a high-level hardware compiler called *Flamel*. Ordinary Pascal programs are used to define the behavior required of the hardware. Flamel undertakes to find parallelism in the program, so it can produce a fast-running implementation that meets a user-specified cost bound.

A number of program transformations create sections of code with more parallel computations than the original program has. A novel feature of Flamel is a method for organizing the search for the transformations that best satisfy the goal. Another new algorithm is one for "expression height reduction": rewriting an ensemble of expressions using algebraic properties in order to compute the expressions faster.

An implementation of Flamel has been completed. The output is a description of a datapath and a controller, and at a sufficient level of detail so that good area and execution time figures can be estimated. On a series of tests, Flamel produces implementations of programs that would run 22 to 200 times faster than an MC68000 running the same programs, if the clock cycles were the same. The tests also show that a wide range of time-area tradeoffs are produced by varying the area constraint.

## I. Introduction

MUCH OF THE early work in hardware synthesis took hardware descriptions that were mostly *structural*, meaning that the input described the architecture of the required system and the synthesis systems instantiated that architecture in some technology. More recently, it has become feasible to build compilers that accept *behavioral* descriptions, where the input simply describes what the required system needs to do. Such compilers need to choose an architecture; after that, the techniques developed for the structural synthesizers can be used. This paper describes a behavioral hardware compiler called Flamel.[1]

A typical way of describing behavior is to write a program in an ordinary computer language, with some convention as to how to communicate with the "outside world." Roughly speaking, the required system needs to act the same way across the communication channel that the program does. The hardware compiler can rearrange the program and have calculations done in parallel, so long as the proper values are written to the outside world at the proper times. Some examples of compilers that operate this way are: the CMU-DA project [1], particularly the Design Automation Assistant [2], [3] portion; Arsenic [4]; the USC Design Automation project [5]; the AT&T Bell Labs VLSI Design Automation Assistant [6]; and SC [7]. There are some similarities between Flamel and these other systems. The thing that distinguishes Flamel from the others is extensive modification of the input program, together with an algorithm for deciding which modifications yield the fastest-executing chips while meeting a user-supplied cost constraint.

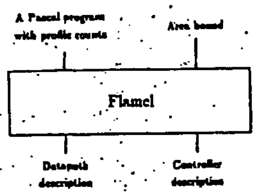


Fig. 1. Flamel's operation.

An overall picture of Flamel's operation is shown in Fig. 1. The user provides a Pascal program together with execution frequency counts for a typical execution of the input program. The other user input is a number saying roughly how much hardware is allowed. The output is a design for hardware that will perform the same function as the Pascal program.

## II. The Compilation Problem

The problem that Flamel attempts to solve is:

> Given a program *P*, find a hardware implementation with the same external behavior as *P*. The implementation should minimize the execution time of the implementation running on typical data, while meeting a user-supplied constraint on the cost of the hardware implementation.

This section will define *hardware implementation* and *external behavior*, and explain how Flamel estimates execution time and hardware cost.

### A. Hardware Model

The general model for a circuit produced by Flamel is that of a synchronous digital machine consisting of a *datapath* and a *controller*. The datapath consists of *functional units* (ALU's, adders, registers, I/O pads, etc.) interconnected by busses. Functional units source, sink, hold, and operate on data words, where a word contains some number of bits. A given bus may receive data from and give data to several functional units. This means that

Manuscript received February 16, 1986; revised November 12, 1986. This work was done while the author was at Stanford University. Supported by DARPA contract N00039-83-C-0136.

The author is with AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.

IEEE Log Number 8612997.

[1]After a medieval alchemist said by some to have achieved the "Great Work" of gold-making.

multiplexers may be needed on functional unit inputs and outputs in order to regulate the bus traffic. Note that this model allows anything on the spectrum between global busses and dedicated connections. Collectively, functional units and busses are called *resources*.

The controller is a finite-state machine. The inputs to the machine come from the datapath's functional units—information like the low-order bit of a register or the status bits of an ALU. The outputs control the datapath, setting the function for multifunction units, and determining multiplexer selector settings.

It is useful to impose a structure on the finite-state machine. Flamel builds a machine consisting of *blocks*, where each block is what would be called "straight-line code" in an ordinary compiler. (Sometimes the term "block" is used in hardware literature to denote a subcircuit, but we don't use the term in that sense. A Flamel block is not usually implemented as a distinct circuit, but rather, as some portion of microcode memory.) There are a fixed number of *cycles* in a block, and a block always executes each cycle in turn so that it finishes some fixed, data-independent time after it is entered. Some of the operations in a cycle might be done only conditionally, so it is not quite like "straight-line code" in that sense. When a block finishes, it passes control to another block (*activates* it), perhaps using a datapath output to choose among several possibilities.

Given a hardware implementation for a program $P$, Flamel tries to optimize the *estimated running time of* $P$, denoted $T(P)$. For a block $B$, let $freq(B)$ be a *frequency count*: an estimate of how many times the block executes in a typical run of the input program. That number is derived from data that the user must supply (described below). Also, let $H(B)$ be the number of cycles required to execute block $B$. Then

$$T(B) \overset{\text{def}}{=} freq(B) \times H(B)$$

and $T(P)$, the time for a whole program, is

$$T(P) \overset{\text{def}}{=} \sum_{B \in bls(P)} T(B)$$

where $bls(P)$ is the set of blocks implementing $P$. Typically, the user runs the (software) program on sample input data, and uses a profiling tool to find out how often each statement executes. If those numbers are used to derive the $freq(B)$ values, then $T(P)$ will be the number of cycles that the Flamel-generated chip will need to execute when presented with the same input data.

The cost of a hardware implementation for a program $P$, denoted $C(P)$, depends on the implementation technology. The prototype Flamel is targeted to a VLSI implementation, and $C(P)$ is an estimate of the chip area. We adopt a layout scheme that has found some success in other projects: both MacPitts [8] and Shrobe's datapath generator [9] used a bit slice architecture. The functional units are arranged in a vertical line and the busses run in between each bit of the units. The busses needn't run the full length of the line—only as far as necessary. Thus, you get the effect of several local busses sharing the same routing track.

Flamel's current output describes a data path by naming the functional units in the order they appear, and the busses with their attachments. Flamel has a table with the height of each possible functional unit (assuming predesigned cells), and other data such as the width of all the cells, and the height and width increments incurred by additional wiring tracks. From that, it is possible to make a very good estimate of the area of a bit slice. The output also describes a controller, by giving the states and transitions of a finite-state machine. There are other programs that generate VLSI layouts for the controllers (SLIM [10], and a *regular expression compiler* [11]), but the prototype Flamel doesn't get feedback on how big the controllers might be, so the controller area isn't included in its cost estimate.

## B. Pascal as a Behavior Specification Language

The input to Flamel is a Pascal program. Suppose that a statement like "read$(x)$" is executed at some point during the running of the input program. Then the hardware implementation should have an input port with enough pads to read all the bits of $x$; a value for $x$ will be read at some point during the running of the chip. Similarly, a statement like "write$(x)$" implies an output port in the hardware implementation.

A hardware implementation of a Pascal program will be *valid* if the following is true:

> Given any set of input data, the program and the hardware implementation must do I/O in the same order, and with the same values.

It is convenient to allow more than one value to be read or written in a single cycle. Flamel uses this convention: a statement like "read$(x, y)$" means that the hardware implementation should read $x$ and $y$ at the same time. Thus, there need to be two input ports in such a case. For reads that aren't to occur simultaneously, only one port need be provided (in analogy to one input stream for a software program). A similar convention holds for writes.

The prototype Flamel imposes some restrictions on the input programs.

- The program must be a single, parameterless, nonrecursive procedure.
- The only datatypes allowed are: integer, boolean, char, and subranges and one-dimensional arrays of these.
- Multiplication, division, and mod are not allowed, except by constants that are powers of 2.

Only the first restriction requires much of an extension to the techniques developed for Flamel.

Flamel converts the input program into an internal form, dubbed *dacon* (the dataflow/control-flow). Some properties of dacons need to be explained before Flamel's algorithms can be understood.

The dacon representation of a program is a combination of two data structures commonly used in software compilers [12]. The program is divided up into *basic blocks*: sections of code with no jumps into or out of them except at the beginning and end. There is a *block graph* (often called a *flow graph*), whose nodes represent blocks and whose arcs represent the "transfers-control-to" relation. For each block, there is a *directed acyclic graph* (DAG) that gives the detailed calculations of the block. The nodes of the DAG are operators (*add, constant source, pad read, array element read*, etc.) and the edges carry values.

The reason for the name "dacon" is that the representation implies a hardware implementation where the blocks operate in a dataflow-like manner (i.e., calculations can be done as soon as input values are ready), but control passes between blocks as in ordinary programs.

One important feature of the dacon representation of a program is that the local nonarray variables of a procedure are not assumed to reside in fixed places. Instead, all the needed variables are passed from block to block. Some of the DAG leaves are *input shadow* nodes, representing the places where needed values are to be found. Some of the DAG roots are *output shadow* nodes, representing places where the values needed by the following block(s) are to be put. A block only has to pass a variable's value on if that value is *live* (potentially used by a direct or indirect successor block). Treating variables this way provides automatic register allocation, when combined with Flamel's resource allocation mechanism.

A complication with DAG's is the need to deal with execution order constraints. For instance, if the original source program has one read statement and then another, we are supposed to ensure that these happen in the hardware implementation. Similarly, a write to an array element $a[i]$ and either a read or write of $a[j]$ must occur in the same order as in the source program, unless we know that the subscripts can't be equal. The method for enforcing such constraints is to use special DAG edges called *control edges*. A control edge from node $x$ to node $y$ means that node $x$ must execute before node $y$ in any hardware implementation of the block.

There are many similarities between the dacon data structure and the "Value Trace" used in the CMU-DA system [13]. Dacons have a better separation between control-flow and dataflow, and that seems to simplify the global optimization process. But the Value Trace is close enough that it could probably have been used instead.

## III. Choosing the Block Structure

The initial dacon for a program is built by dividing the program into basic blocks—sections of code that are always entered at the beginning and exit only at the end. Now if you look at the calculations going on inside such a block, you find that sometimes there are operations that can be done simultaneously because neither depends on

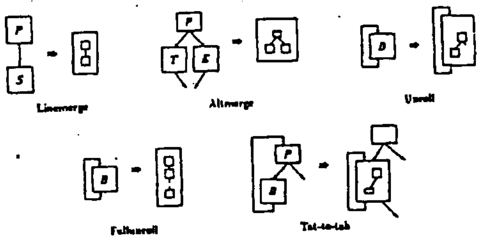


Fig. 2. Flamel's block-level transforms.

the other, directly or indirectly, for inputs. Such operations constitute potential parallelism.

People have done empirical studies of the average amount of potential parallelism in a basic block of a typical high-level language program. It turns out that there isn't very much: typically, the studies show that the average "degree of parallelism" (number of operations done at any given time) is at most two or three [14]. Thus, a compiler that exploits only basic block parallelism isn't going to be able to produce a wide range of time/area tradeoff implementations.

Flamel's way out of this problem is to transform the dacon to make bigger blocks with more calculations in them. This turns out to yield impressive amounts of parallelism—usually a parallelism degree of five to ten.

### A. Flamel's Block-Level Transforms

The five block-level transforms in Flamel's repertoire are shown in Fig. 2. Each creates a single block whose DAG is formed in a simple way from the DAG's of the original block(s).

*Linemerge:* Connect the outputs of $P$ to the inputs of $S$.

*Altmerge* (Alternative merge): Attach both $T$ and $E$ to the outputs of $P$, and then use multiplexers controlled by $P$'s exit condition to choose the correct outputs to pass on to the common successor of $T$ and $E$. If $T$ or $E$ have nodes with side effects (e.g., writing an array), those nodes need to be flagged so that they execute only when they should. There is also a variant where $T$ or $E$ is missing.

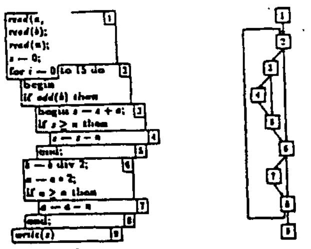
*Unroll:* Attach a copy of $B$ to the outputs that lead to restarting the loop, and use multiplexers to choose the correct outputs (from the first or second body) to pass on to the loop's successor.

*Fullunroll:* Like unroll, but use as many copies as there are loop iterations (when that number is known at compile time).

*Tat-to-tab:* A "test-at-top" loop has two blocks, the top one testing to see if the bottom—the body—should execute. The unroll transforms only work on the one-block "test-at-bottom" loop, so this transform creates one by incorporating the test block at the end of the body block.

It is possible to define a formal notion of *computational equivalence* of dacons, and show that the above transformations preserve it [15].

As an example, consider the *mmult* procedure, which calculates *ab* mod *n*, given *a*, *b*, and *n*. Fig. 3 shows the code and the initial block graph; the boxes surrounding sections of code show approximately what happens in each block. The first transformation on Fig. 3 might be an altmerge of blocks 3 and 4. That would produce a block that calculates $s + a$ and $s + a - n$ and passes one or the other along, depending on the concurrently calculated $s + a \geq n$. Then the newly-formed block could be linemerged with block 5. Further transformations could eventually produce one single block for the whole program.



```
read(a);
read(b);
read(n);
s ← 0;
for i ← 0 to 15 do
begin
if odd(b) then
begin s ← s + a;
if s ≥ n then
s ← s − n
end;
b ← b div 2;
a ← a * 2;
if a ≥ n then
a ← a − n
end;
write(s)
```

Fig. 3. *mmult* and block graph.

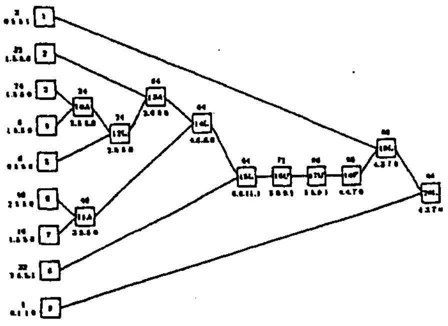


Fig. 4. Transform tree for *mmult*.

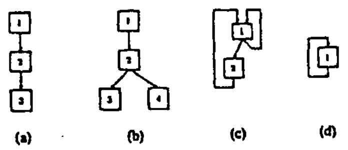


Fig. 5. Ambiguous cases for transforms.

## B. The Transform Tree

The transformations introduced in the previous subsection are not particularly new. What *is* new in Flamel is a controlled way for deciding which transforms to apply. Central to the method is a data structure called a *transform tree*. The *transform tree* has the program's original basic blocks as leaves, and the internal nodes represent blocks that were formed during the transformation process. They are labeled with letters to say how they were formed from their children. For example, Fig. 4 shows the transform tree for *mmult*. (For the moment, ignore the numbers surrounding the blocks.)

The significance of the transform tree is that it succinctly represents the search space Flamel explores when it tries to decide on the block graph of the dacon it will implement. Conceptually, Flamel chooses the blocks to implement as follows:

> Either choose the root block to implement, or choose implementations for its children (recursively) and connect them as dictated by the transformation-type label.

Choosing the implementation this way is very attractive, for two reasons. First, the total number of blocks examined is small—about twice the number of original blocks. And second, it avoids "local optimum" problems, where one stops transforming because any single extra transform worsens things, even though a worsening transform might be necessary to reach a global optimum.

For the most part, there is only one transform that can be done involving any given block. There are four exceptions to this, shown in Fig. 5. Flamel chooses the linemerge 1-2 in case (a), the altmerge in cases (b) and (c), and several unrolls before a fullunroll in case (d). The important case is (b); the choice there is based on the experience that a linemerge doesn't often increase the amount of parallelism, since the critical computation path in the predecessor block often connects to the critical path in the successor. At any rate, the linemerge can be done after the altmerge, resulting in the same blocks as if the transforms were done in the other order, so the disambiguation choice doesn't have a huge effect on the final transform tree.

A problem is that, early in the transformation process, the graph may not look like Fig. 5(b), yet it may after some portion of the graph collapses into one block. To avoid doing the linemerge too early, Flamel only does linemerges when none of the other transformations apply to the second block. Flamel builds the transform tree assuming that fullunrolls can always be done, postponing the actual building of the DAG's for the transformed blocks. This tactic means that any dacon derived from a Pascal procedure that doesn't use goto's or case statements will be reduced to a single block, as evidenced by the relation between the Pascal compound statement forms and the transform patterns. Procedures that have goto's can still be handled by Flamel, but there may be less opportunity to find parallelism.

## C. Choosing the Blocks

After the transform tree has been made, the next step is to pick the set of nodes that will be used in the dacon to be implemented. Recall that the goal is to find an approximate solution to the *mintime*$(P, c)$ problem—that of minimizing execution time while meeting cost constraint $c$. The *mintime*$(P, c)$ problem seems to be extremely difficult to handle, because each block will have its own cost-time tradeoffs. Flamel's method is to solve a simpler problem, called *rmintime*. Instead of meeting a cost bound, the implementation must meet a *resource* bound:

$$rmintime(P, R) \overset{\text{def}}{=} \min T(P), \quad reses(P) \subseteq R$$

where $R$ is a set of resources, *reses*$(P)$ is the set of resources used in the implementation of $P$, and the minimization is over all possible implementations. So a typical *rmintime* problem might be: find a minimum time implementation of program $P$ that uses three ALU's, five registers, eight constants, and two I/O ports.

The great thing about the *rmintime* problem as opposed to the *mintime* one is that each block can be chosen independently. Since the blocks don't overlap execution,[2] resources can be reused from block to block, so that if each block meets the resource bound independently, then the whole program will.

Now Flamel's block-choosing method can be presented.

1. Use the cost bound $c$ to derive a resource bound $R$. Flamel's current method is to allocate approximately equal portions of $c$ to each of the resource classes: ALU, register, constant, bus. A typical cost for each of these resources is estimated and divided into the cost portion to get the number of each in $R$. The number of I/O pads and register files allowed is dictated by the program (we have no choice).

2. Build DAG's for the blocks represented by nodes in the transform tree. Use a method to be presented later to solve *rmintime*$(B, R)$ for each block $B$ and yield cost and time estimates for each block..

The DAG building proceeds bottom-up, using a postorder traversal of the transform tree. Don't bother building a DAG if the children blocks exceed the resource bound $R$, since such a DAG will almost certainly also exceed the resource bound. Also, a block may be unimplementable because it is a fullunroll of a loop that cannot be fully unrolled.

3. For each block $B$ that meets the resource bound, calculate

$$besttime(B) = \begin{cases} T(B) & \text{if } B \text{ is a leaf} \\ \min\left(T(B), \sum_{C \text{ child of } B} besttime(C)\right) & \text{otherwise.} \end{cases}$$

This number represents the smallest total amount of time that will be spent in block $B$ if the fastest implementation is chosen. The "fastest implementation" uses either the DAG for $B$, or the fastest implementation (defined recursively) of its children. Let $T(B) = \infty$ if $B$ doesn't meet the resource bound.

4. Finally, reconstruct the dacon that yields the fastest implementation. Do this with a preorder traversal of the dacon that was left after building the transform tree (probably a single block). When visiting a node $B$, see if *besttime*$(B) = T(B)$; if so, $B$ should be part of the chosen dacon, so return without visiting the children. Otherwise, *undo* the transform that led to $B$: remove $B$ from the current block graph and replace it with its children, interconnected as dictated by the transform type.

Fig. 4 gave the transform tree for *mmult*, derived by using the rules just given. If Flamel is given a large resource bound, then all of the blocks have DAG's built for them, and Fig. 4 shows the $T(B)$ value[3] (on top) and estimated resource set (on bottom) for each node. The resource set is given by a string of the form $w.x.y.z$, where $w$, $x$, $y$, and $z$ are the numbers of ALU's, registers, busses, and constants, respectively. For example, block 3 has a DAG that can be executed in three cycles, and thus $T(B_3) = freq(B_3) \times H(B_3) = 8 \times 3 = 24$; and, Flamel estimates that one ALU, five registers, and five busses suffice to implement it.

Many people have proposed program transformation systems without a global strategy for deciding which transforms to apply. Such systems would probably get stuck in locally optimal configurations. In the *mmult* example, the 2–12L altmerge yields a slower block, but doing it enables the other transformations, yielding a final implementation consisting of the single block 20L. The final implementation chosen by Flamel will execute in 66 cycles versus 140 for the locally optimal implementation (blocks 1, 2, 12L, 11A, 8, 9).

The user can supply a cost bound, nominally the height of the bit slice, and Flamel derives resource bounds from that. A bound allowing only two ALU-class functional units will cause Flamel to omit trying to implement blocks 15–20, and the best-time block set satisfying the bounds turns out to be the locally optimal one mentioned earlier. Note that the cost bound only loosely defines the architecture. Flamel's resource allocation procedure chooses two adders, two shifters, and one ander for this example—it turns out to be cheaper than two ALU's.

## IV. DAG Transforms

Flamel needs as a subroutine a method for solving the *rmintime* problem for blocks. Recall that the *rmintime* problem is to minimize the $T(P)$, the expected amount of time spent executing a program. For a block $B$, $T(B)$ is the product of the estimated block frequency and the number of cycles it takes to execute the block, $H(B)$. When working on a single block, we can equally well say that the *rmintime* problem is the minimize $H(B)$.

[2]At least, not in the prototype Flamel.

[3]The frequency values assume that the If tests are true half of the time.

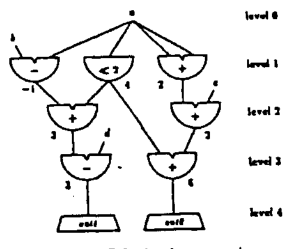


Fig. 6. DAG for level compression.

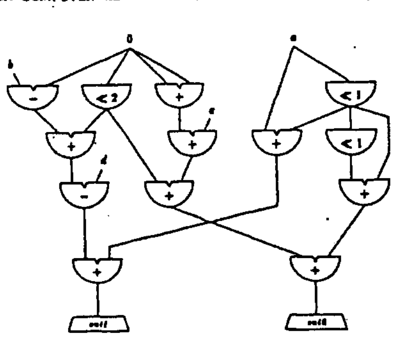


Fig. 7. DAG after level compression.

More precisely, the problem is to take a DAG and decide in which cycle each node is to start performing its computation. The *schedule* needs to satisfy certain properties: a node must be scheduled no earlier than some specified delay after each of its predecessors, and the resource constraints have to be respected in each cycle. A further aspect to the problem is that certain transformations may be applied to the DAG, transformations allowed by the algebraic properties of the operators involved. This process is called *height reduction*, since the number of cycles used is the DAG height when all operators use one cycle to complete.

## A. Height Reduction

The problem of reducing DAG height has been much studied in the special case where the DAG is a tree and the operator nodes are +, −, ×, and /. See Chapter 2 of Kuck's book [16] for a survey. Unfortunately, the restriction to trees makes those algorithms essentially useless for Flamel. Height reduction is only effective on large graphs, and the only large graphs created by Flamel are those created by loop unrolling. The DAG's of unrolled loops can be turned into trees by copying shared subexpressions, but they usually grow exponentially if that is done. Another problem with the published literature on height reduction is that it doesn't say how to deal with multiplexers.

Flamel incorporates a new height reduction algorithm, called *level compression*. The idea is to repeatedly pick some node in the DAG and shorten the distance between that node and the roots (sinks). Figs. 6 and 7 show how a node is *moved later*.[4] The numbers near the nodes in Fig. 6 are *contribution factors with respect to a*: informally, if an algebraic expression were written for the value calculated by a node $g$, the variable $a$ would appear multiplied by the contribution factor of $g$. (Actually, contribution factors are defined inductively by saying that $a$ has a contribution factor of 1, and giving rules of propagating contribution factors through nodes [15].) The move is accomplished by replacing $a$ with 0 and adding compensating multiples of $a$ according to the contribution factors at some level as late as possible. Fig. 7 shows the result; there is no height reduction here, but there would be in a larger example, such as a block arising from loop unrolling. Flamel performs *constant folding*—simplifications due to some or all inputs to a node being constant—yielding a much simpler DAG from Fig. 7.

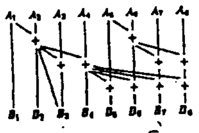
The method for choosing the node to move is basically: look for a node about halfway along a critical path (a path where all the time constraints are tight). As height reduction proceeds, certain nodes are *frozen* to prevent them from taking part in further reductions: nodes created for contribution calculations, nodes that contributions are added to, and nodes on paths from frozen nodes to sinks. Then nodes chosen for later moving must be about halfway along nonfrozen portions of critical paths. Without this freezing mechanism, the height reduction procedure usually cannot do better than approximately halve the original height. The moving process stops when there is no further height reduction or when the resource bounds are exceeded at some level. Care has to be taken to avoid doing transformations that increase the DAG height. For details, see Trickey [15].

One of the central methods of many parallelizing compilers is to examine loops to see if they have a special form, and then use one of a library of transformations to convert the loop into a more parallel form. For example, a *parallel prefix sum* calculation like

$$\text{for } i \leftarrow 2 \text{ to } n \text{ do } a[i] \leftarrow a[i-1] + b[i]$$

has the form of a *linear recurrence*, and a technique such as Kogge and Stone's [17] can calculate all of the $a[i]$ in $O(\log n)$ time. If the loop contains an if statement, the method of Banerjee *et al.* [18] might be used. Flamel takes a different approach: rather than looking for special cases, all loop parallelization is done by loop unrolling followed by DAG height reduction. This approach yields the fastest possible calculation of the above loop, and it has the advantage that it would also work on a partially unrolled loop (for less parallelization when the cost bound dictates). It would also work if there were other calculations in the loop that were not in linear recurrence form, if those other calculations were not the bottleneck. Fig. 8 shows

[4]The nodes labelled "<< n" mean "left shift n bits."



$A_1$ $A_2$ $A_3$ $A_4$ $A_5$ $A_6$ $A_7$ $A_8$
+
$B_1$ $B_2$ $B_3$ $B_4$ $B_5$ $B_6$ $B_7$ $B_8$

Fig. 8. Level compression height reduction on parallel prefix sum.

the result of level compression height reduction of the fully unrolled parallel prefix sum loop.

## V. Hardware and Control Generation

After Flamel has chosen the dacon to implement, it needs to tackle the job of actually scheduling the nodes in all of the DAG's and assigning hardware resources to implement the operations and value transfers. Registers are resources, so register allocation occurs here too. In the prototype Flamel, only registers can hold values for more than one cycle, so registers are sometimes needed to hold intermediate values.

Flamel uses a method called *folding*, similar to a method used by Hitchcock and Thomas [19]. The basic idea is to start out with the *earliest possible* schedule and a resource assignment that gives every node its own functional unit and every edge its own bus. Then, pairs of resources that perform the same function, or could be generalized to perform the same function, are *folded* together into one resource if they aren't being used simultaneously. When there is a choice of pairs to fold, the pair yielding the best expected improvement in the hardware cost is chosen.

Flamel goes a step further than Hitchcock and Thomas. If there are no more possible folds, yet the estimated hardware cost is still more than the user allows, then Flamel *lengthens* the schedule: it chooses a pair of resources that could be folded if certain DAG operations didn't overlap in time, and then it forces those operations into different cycles by adding judicious control edges and rescheduling.

For an example of this resource allocation procedure consider the *mag* program shown in Fig. 9. This is a program for approximating $\sqrt{a^2 + b^2}$, translated from the MacPitts input given by Southard [20]. A comment follows the first statement of each basic block, giving the "frequency" fed to Flamel for that block. With no resource limit, Flamel chooses a single-block implementation for *mag*, with the DAG shown in Fig. 10. The rectangles with round ends are registers, added because none of the functional units have output buffers. Some nodes have multiplexers on their inputs (marked by lines across the top): the left or right input is used depending on whether the value shown feeding the end is 1 or 0, respectively. Each level of Fig. 10 will be a different cycle during execution. Notice how the two separate read statements have forced the pad reads (*in* nodes) to be on different levels.

Given the DAG of Fig. 10, the resource allocation procedure comes up with the hardware resources shown in Fig. 11: input pad, three registers, negater, adder/subtractor/comparitor, constant source, barrel shifter, output pad, and four busses. Again, input multiplexers are shown as lines across the top, with choices controlled via control lines (not shown). As an example of folding, the *aab*, *bab*, *g*, and *sqs* registers of Fig. 10 have all been folded into the *other* register of Fig. 11, but Flamel's cost tables said it was cheaper to leave *a* and *b* in separate registers (connections to several busses are almost as expensive as registers). The DAG edges out of the pads, *aab*, *b*, and *g* were folded into the leftmost bus of Fig. 11 and the edges out of *bab*, *sqs*, and '−' were folded into the rightmost bus. Those bus connections helped determine that it was free to use the *other* register for *sqs*.

The final step is to generate description files for the datapath and controller. Flamel uses an adaptation of Kernighan and Lin's heuristic procedure for partitioning graphs to choose the top-to-bottom ordering of the functional units in a bitslice [21]. The goal is to allow the

```
procedure mag;
  var a, b, aab, bab, g, l, sqs, res: integer;
  begin   {1}
  read(a);
  read(b);
  if (a < 0) then aab ← −a   {.5}
  else aab ← a;   {.5}
  if (b < 0) then   {1}
     bab ← −b   {.5}
  else bab ← b;   {.5}
  if (aab > bab) then   {1}
     begin   {.5}
     g ← aab; l ← bab
     end
  else begin   {.5}
     l ← aab; g ← bab
     end;
  sqs ← (g − g div 8) + l div 2;   {1}
  if (g > sqs) then res ← g   {.5}
  else res ← sqs;   {.5}
  write(res)   {1}
  end;
```

Fig. 9. Flamel input for magnitude approximation.

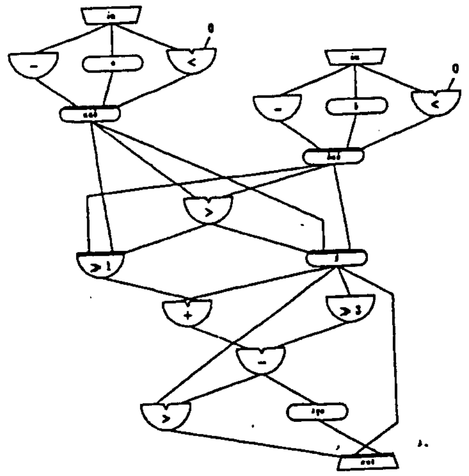


Fig. 10. DAG chosen to implement *mag*.

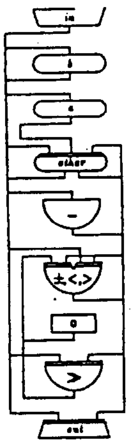


Fig. 11. Bitslice for *mag* datapath.

maximum amount of sharing of wiring tracks for busses. In the *mag* example, the ordering allows two short busses to share the track second from the left, and then keeps the rightmost bus as short as possible. The actual output isn't particularly interesting to look at. The datapath description file lists the resources in order, along with their particular capabilities and I/O multiplexers, and then the connections between busses and functional units. The controller description file lists for each block, cycle-by-cycle, what values are gated to and from what busses, and what the functional units do.

## VI. Results

It is hard to characterize theoretically what a "typical" input program looks like, so theoretical results about Flamel's algorithms are hard to come by. For this reason, a prototype implementation has been produced.

How should the quality of Flamel's output be assessed? One way is to compare it with the output of other silicon compilers. Nearly all of the previous work similar to Flamel is not yet at the stage where results are being reported. The main exception to this lack of results is the work at CMU [1], [2]. Unfortunately, their test cases are things like general-purpose computer architectures; Flamel would certainly do rather poorly on such tests, since no attention has been paid to the details that are important for such tests. In particular, the poor support for bit manipulation in Flamel is fatal. Also, the programs for emulating general-purpose computers tend to be uninteresting from the point of view of global optimization, since the instruction fetch-decode-execute loop takes only a few cycles.

One thing is certain: a chip produced by Flamel better run considerably faster than an off-the-shelf microprocessor running the input program, because the microprocessor will probably be cheaper, owing to economy-of-scale. At the time this research was begun, the Motorola MC68000 was a typical microprocessor, and the availability of a good Pascal compiler targeted to the M68000 made it a natural microprocessor to compare with.

TABLE I
Speedups: MC68000 External Clock Cycles/Flamel Chip Cycles

| | Most Serial | Basic Blocks | Most Parallel |
|---|---|---|---|
| minimum | 8.3 | 9.9 | 22.7 |
| maximum | 14.5 | 20.4 | 217.3 |
| median | 11.1 | 15.1 | 43.5 |
| average | 11.7 | 15.9 | 67.8 |

Each of the test programs was compiled with pc68, a Stanford-produced Pascal compiler that generates good code. It incorporates Chow's global optimizer [22]. The assembly listing of the generated code for each test was examined to calculate how many external clock periods an MC68000 would use in executing the program, given the same basic block frequencies that Flamel used in calculating execution time.

Some statistics about the ratio of MC68000 cycles to Flamel cycles are shown in Table I. The numbers show how many times faster a Flamel chip would be compared to an MC68000 if they were both clocked at the same rate. Of course, the MC68000 could probably be clocked faster than a Flamel chip because it was hand-tuned by experts. But a Flamel cycle need only be long enough to do a bus transfer followed by an add, so hand tuning could probably make a Flamel chip operate at a rate comparable to the MC68000. Even if the Flamel chip cycle were twice as long as that of the MC68000, the speedups are still good.

There were 15 test programs, chosen to be typical of code fragments that might appear in general-purpose software. The programs do things like bubble-sort, convert strings of digit characters to numbers, count words, implement a finite impulse response filter, and manage a hash table. The hash table program was the longest, about 70 lines. Complete listings can be found in Trickey [15].

The "most serial" column of Table I is what Flamel produced when asked to put only one noncopy operation in each cycle and not to do any block-level transformations. The "basic blocks" column is the most parallel possible implementation without doing block-level transformations. The "most parallel" column gives the fastest implementation Flamel could find.

One can see from the "most serial" column that an order of magnitude speedup is due simply to the fact that a Flamel chip doesn't have to fetch instructions and data from memory, and doesn't have to do address arithmetic. But the difference between the "most serial" and "most parallel" columns shows that when all of Flamel's techniques are brought into play, there are an average of 4.8 noncopy operations being done in parallel. The "most serial" implementation is similar to what would be achieved using a microcoded standard processor. An experiment

TABLE II
DATA FOR SOME BENCHMARK IMPLEMENTATIONS

| Test | Cycles | Datapath Area ($\lambda^2 \times 10^6$) | Controller Area ($\lambda^2 \times 10^6$) | Compilation Time (s) |
|---|---|---|---|---|
| presum | 58 | .850 | .068 | 4.7 |
| | 30 | 1.011 | .062 | 4.7 |
| | 22 | 1.373 | .180 | 15.6 |
| | 12 | 1.463 | .138 | 14.5 |
| | 6 | 8.342 | .090 | 16.5 |
| | 8 | 8.342 | .096 | 21.1 |
| mag | 19 | .759 | .160 | 18.4 |
| | 13 | .876 | .104 | 19.6 |
| | 9 | .897 | .082 | 18.1 |
| mmult | 204 | 1.063 | .210 | 21.2 |
| | 100 | 2.029 | .165 | 16.9 |
| | 68 | 2.796 | .167 | 17.9 |
| | 64 | 2.514 | 1.267 | 463.3 |
| hash | 444 | 8.716 | 1.042 | 106.6 |
| | 246 | 10.171 | 1.067 | 103.1 |
| | 167 | 18.729 | 3.644 | 617.4 |

reported in Trickey [15] showed that a microcoded RISC architecture (MIPS [23]) would run in about as many cycles as a "most serial" Flamel chip.

One of the most interesting results of these tests is that there is hardly any parallelism available if no block-level transforms are done. Some previous attempts to parallelize ordinary programs have met with discouraging results, but those attempts didn't look for parallelism at more than a basic block at a time.

Let us now examine four of the benchmarks in detail, to see what time/area tradeoffs can be achieved. Table II contains data about some of the implementations that Flamel finds with different settings of the cost bound. *Mmult* and *mag* were given in Figs. 3 and 9, *presum* does the parallel prefix sum computation discussed at the end of Section IV, and *hash* is an expanded version of the hash table procedure in Kernighan and Plauger [24]. The areas are given in units of mega-$\lambda^2$, where $\lambda$ is half the minimum feature size. The nMOS cells used in the area calculations are similar to those used in Stanford's Information Systems Laboratory [25].

All of the *presum* implementations are given. It can be seen that Flamel achieves a wide range of time/area tradeoffs. The slowest implementation has a lengthened schedule in an attempt to meet the cost bound, and the fastest implementation uses a fully unrolled loop, height reduced. The intermediate cases had partial degrees of block transforms, loop unrolling, and height reduction. The table shows only some of the implementations for the other tests. The *mag* designs are similar to or better than those found by MacPitts, but MacPitts requires that the user change the input to get the different designs [20]. The author implemented *mmult* by hand; it was several times smaller than the comparable Flamel design, but the one implementation took several months to complete.

Another thing to notice from Table II is that the compilation time is reasonable. The times are given in CPU seconds on a VAX 11/780. Furthermore, Flamel would run much faster with more sophisticated data structures.

The areas estimated for the most parallel versions of the test programs were almost all smaller than today's VLSI microprocessors. The exceptions were programs that could be made faster by making large, many-ported register files to implement some arrays. Much of the area required by *hash* is due to seven arrays, with a total of 343 entries. In most of those cases, tapped shift registers could have achieved comparable speedups at a fraction of the cost. A future version of Flamel should know how to use shift registers. If larger programs than the hash table manipulator were tried, the datapath area probably wouldn't grow too much unless more and larger arrays were used. Larger programs probably won't be able to have much more than the approximately five things going on at once that Flamel found in the test programs. On the other hand, the controller area would start to dominate if much bigger programs were tried. Flamel has to pay more attention to controller area than it does.

Some tests were run to see which of Flamel's techniques were most important in getting best speedup over the most serial implementation. Here are the average percentages of total speedup achieved when certain transformations were omitted:

- block transforms omitted: 26%,
- loop unrolling omitted: 40%,
- height reduction omitted: 71%.

See Trickey [15] for further statistics, including statistics about the efficacy of the cleanup transformations.

## VII. CONCLUSIONS

The results of running Flamel on test cases are encouraging. Respectable speedups can be found when a datapath with multiple functional units is used rather than a more standard architecture with a single ALU. Also, Flamel has shown itself capable of producing a wide range of implementations, so a user can quickly get many designs with different performance-area tradeoffs. While the emphasis in this work has been on custom VLSI for the implementation target, most of the techniques would apply equally well to gate array and standard cell implementations.

Compile time is the best time to schedule, when it works. This argument has been put forward by others [26], [27] as an answer to dataflow machines. A dataflow machine can be viewed as determining at execution time the operations that can be done in parallel, and scheduling them on available processing units. It also has to schedule the transfer of data to and from memory. All of this work is expensive in hardware. Furthermore, it appears to slow the rate at which individual processing elements can be supplied with operands. The result is that a lot of parallelism needs to be found in the executing programs before the dataflow machine beats a conventional processor.

Some parallelism can only be detected at execution time. For example, two array subscripts might depend on input data, so that one has to wait until execution time to determine whether certain calculations can be done simultaneously. It remains to be an red how much of this type of parallelism can be expected, but the results reported elsewhere [28], [29] show that at least a 5–10 times speedup can be expected from scientific programs, and the results of this work show that a 2–8 times speedup can be expected from general-purpose software.

Some of the contributions of this work are as follows.

- It has been shown that one easy-to-write behavior specification can generate a wide range of time/area tradeoff implementations, with a bare minimum of effort required from the user. Up to now, general-purpose digital system design has been regarded as a domain where an expert is needed to either choose the architecture or at least to guide the choice.
- An adequate set of global (block-level) and local (DAG-level) transformations has been identified, along with a controlled method for applying them. It is easy to suggest optimizations to apply during silicon compilation—after all, most of them have long been known to the software compiler community. Others have done this, but none have given feasible methods for deciding which to apply and when. For instance, Kowalski's system incorporates a number of global optimizations, but the user has to decide by hand which to apply and when [2]. Flamel's dacon-choosing procedure is a step in that direction. Also, the optimizations described here have all been found to be useful in practice, since they were added on an as-needed basis.
- This is the first work to do extensive global optimizations at all; yet the results show that the global optimizations are crucial for extracting much parallelism from ordinary programs. (Fisher's trace scheduling [29] achieves a similar effect by scheduling at a global level; but he intends it for a fixed architecture).
- A height-reduction technique has been developed to work on DAG's rather than simply trees. The combination of the height-reduction technique and the other transforms yields results that subsume a number of specialized techniques used in vectorizing compilers.
- The prototype implementation revealed useful statistics on the relative cost and effectiveness of the proposed optimization techniques. For instance, it showed what size and speed might be expected from a Flamel-designed chip for various pieces of code. Also, the fact that Flamel runs reasonably quickly shows that a moderate amount of search (in dacon-choosing) is not out of the question.

There is a lost of room for improvement in Flamel. Trickey [15] discusses some ideas in a number of areas that deserve further investigation: 1) more than one procedure in the input program; 2) handling the timing of communication with the outside world; 3) pipelining, both at the operation level and the block level; 4) special implementations for arrays; 5) incorporating controller area into the costs.

## REFERENCES

[1] A. C. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Lieve, and J. Kim, "The CMU design automation system," in *ACM IEEE 16th Des utomation Conf. Proc.* (San Diego), 1979, pp. 73–80.

[2] T. J. Kowalski, "The VLSI design automation assistant: A knowledge-based expert system," Ph.D. thesis, Carnegie Mellon Univ., 1984. Available as CMU Technical Report CMUCAD-84-29.

[3] T. J. Kowalski and D. E. Thomas, "The VLSI design automation assistant: Prototype system," in *ACM IEEE 20th Design Automation Conf. Proc.* (Miami), 1983, pp. 479–483.

[4] K. Palem, D. S. Fussel, and A. J. Welch, "High-level optimization in a silicon compiler," Tech. Rep. TR-215, Dept. of Computer Sciences, University of Texas, Austin, TX, 1982.

[5] D. Knapp, J. Granacki, and A. C. Parker, "An expert synthesis system," in *Proc. of the Int. Conf. on Computer Aided Design*, ACM and IEEE, 1983, pp. 419–424.

[6] T. J. Kowalski, D. J. Geiger, W. H. Wolf, and W. Fichtner, "The VLSI design automation assistant: From algorithms to silicon," *IEEE Design and Test*, pp. 33–43, Aug. 1985.

[7] P. E. Agre, "Designing a high-level silicon compiler," in *Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers* (Port Chester, NY), 1983, p. 413.

[8] J. M. Siskind, J. R. Southard, and K. W. Crouch, "Generating custom high performance VLSI designs from succinct algorithmic descriptions," in *Proc. Conf. on Advanced Research in VLSI*, (MIT), Jan. 1982, pp. 28–39.

[9] H. E. Shrobe, "The data path generator," in *Proc. Conf. on Advanced Research in VLSI*, (MIT), Jan. 1982, pp. 175–181.

[10] J. L. Hennessey, "SLIM: A simulation and implementation language for VLSI microcode," *Lambda*, pp. 20–28, Apr. 1981.

[11] A. R. Karlin, H. Trickey, and J. D. Ullman, "Experience with a regular expression compiler," in *Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers* (Port Chester, NY), 1983, pp. 656–665.

[12] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1977.

[13] M. C. McFarland, "The VT: A database for automated digital design," Tech. Rep. DRC-01-4-80, Design Research Center, Carnegie Mellon University, 1978.

[14] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.*, vol. C-19, no. 10, pp. 889–895, 1970.

[15] H. Trickey, "Compiling Pascal programs into silicon," Ph.D. thesis, Stanford Univ., July 1985. Stanford Computer Science Report STAN-CS-85-1059.

[16] D. J. Kuck, *The Structure of Computers and Computations*, vol. I. New York: Wiley, 1978.

[17] P. M. Kogge and H. S. Stone, "A parallel algorithm for efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, pp. 786–792, Aug. 1973.

[18] U. Banerjee, D. Gajski, and D. J. Kuck, "Array machine control units for loops containing IFs," in *Proc. 1980 IEEE Int. Conf. on Parallel Processing*, 1980, pp. 23–36.

[19] C. Y. Hitchcock III and D. E. Thomas, "A method of automatic data path synthesis," in *ACM IEEE 20th Design Automation Conf. Proc.* (Miami), 1983, pp. 484–489.

[20] J. R. Southard, "MacPitts: An approach to silicon compilation," *Computer*, vol. 16, pp. 74–82, Dec. 1983.

[21] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.

[22] F. Chow, "A portable machine-independent global optimizer—Design and measurements," Ph.D. thesis, Stanford Univ., Dec. 1983.

[23] S. Przybylski, T. Gross, J. Hennessy, N. Jouppi, and C. Rowen, "Organization and VLSI implementation of MIPS," *J. of VLSI and Computer Systems*, vol. 1, Spring 1984. Available as Tech. Rep. 83-259, CSL, Stanford.

[24] B. W. Kernighan and P. J. Plauger, *Software Tools in Pascal*. Reading, MA: Addison-Wesley, 1981.

[25] J. Newkirk, R. Mathews, J. Redford, and C. Burns, "Stanford nMOS cell library," Tech. Rep. 001, Information Systems Laboratory, Stanford Univ., 1981.

[26] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, "A second opinion on data flow machines," *Computer*, vol. 15, pp. 58–69, Feb. 1982.

[27] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel processing: A smart compiler and a dumb machine," in *Proc. ACM SIGPLAN Symp. on Compiler Construction* (Montreal, Canada), 1984, pp. 37–47.

[28] D. J. Kuck, "Measurements of parallelism in ordinary FORTRAN programs," *Computer*, vol. 7, pp. 37–46, Jan. 1974.
[29] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, pp. 478–490, July 1981.

**Howard W. Trickey** (S'76–M'85) received the B.A.Sc. and M.A.Sc. degrees from the University of Toronto in 1978 and 1980, and the Ph.D. degree from Stanford University in 1985.

He is now a Member of Technical Staff at AT&T Bell Laboratories in Murray Hill, NJ. His research interests include hardware synthesis, compilation for parallel machines, and text processing and illustration tools.

10

IEEE TRANSACTIONS ON COMPUTERS, VOL. C-18, NO. 7, JULY 1969

# AT-6 Methods Used in an Automatic Logic Design Generator (ALERT)

VINCENT N. TRANS
EXAMINER
ART UNIT 234

THEODORE D. FRIEDMAN, ASSOCIATE MEMBER, IEEE, AND SIH-CHIN YANG

*Abstract*—The ALERT system converts preliminary high-level descriptions of computers into logic. The input to ALERT depicts the architecture of a proposed machine in a form of Iverson notation. As output, the architecture is "compiled" into Boolean equations, which may then be converted into standard computer circuits.

The purpose is to relieve designers of uncreative detail work. Complex structures may be represented conveniently by macro functions, algorithms, multidimensional arrays, and subscripted expressions. Control logic, intermediate registers, and selection and switching mechanisms are automatically supplied and the resulting design is consolidated and simplified by a variety of techniques. Methods used and reasons for those approaches are discussed. To elucidate operation of the system, a sample design is followed through its gradual development as it is processed by the successive steps of ALERT. Nine pages of circuit diagrams were automatically generated from the example and are included in the Appendix.

*Index Terms*—Architecture, automatic logic generation, compiler of computers, design automation, high-level computer description, Iverson notation, structural implementation of algorithms.

DEVELOPMENT of new computers at IBM has to a large extent been automated. The final stages of production are most fully mechanized, including wiring, circuit module identification, documentation, and control of manufacturing equipment [1]. Earlier development work such as checking, simulation, circuit selection, partitioning, and placement has been automated but to a more limited extent [9]. However, the initial task of logic design must still be accomplished by hand before the automated systems can be used. This work, which involves preparation of detailed diagrams or equations to specify the internal structure of a computer, is time-consuming, expensive, and frequently repetitious and uncreative.

The purpose of the ALERT system is to automate much of this preliminary logic design process. ALERT is a logic generator programmed to run on the IBM 7094. It inputs a description of a proposed computer expressed in high-level language. As output, it "compiles" logic designs which carry out the functions specified.

The input to ALERT, called the architecture, describes the overall computer system characteristics, and from this ALERT generates a logic design in the form of Boolean equations. These equations may then be processed by the regular IBM logic automation [9] and design automation [1] programs to produce circuitry and documentation for the desired computer. ALERT is analogous to a conventional compiler program because it translates its input from a high-level algorithmic form into a detailed machine-oriented form. But rather than generating a program as its output, ALERT generates logic designs for building hardware.

The nature and quality of the designs produced by ALERT is discussed elsewhere [5]. In this paper, the programs comprising the ALERT system will be considered.

Fig. 1 lists successive stages in the development of a new computer and identifies automated systems at IBM associated with each activity. With the aid of the ALERT system, the designer may apply automated techniques at an earlier point of the development than is now the case. In conjunction with other automation systems, development may then proceed from the initial stages through to final manufacture on a machine-aided basis. This should reduce costs, decrease development time, and aid checking and documentation.

The problem of automating this preliminary design work involves conversion of a high-level representation of a computer into a more detailed form. Before discussing techniques used, the choice of a high-level design language should be considered.

## The High-Level Language

Traditionally, designers use informal notes, sketches, block diagrams, or natural language such as English to specify a computer before its logic has been designed. After the logic design has been prepared, the design itself may serve as the basic source document, but until the logic has been completed, a formal description is usually not available. A formal language for specifying computers at a high level would not only provide a possible starting point for automated processing, but it would provide an unambiguous source document for the various groups associated with computer development, such as architects, designers, engineers, marketing specialists, and programmers.

A high-level language should be clear, convenient, capable of specifying any feature of a computer, extendable, machine-readable, and it should allow the user to suppress irrelevant or repetitive detail.

Several formal languages have been proposed for the specification of computers at a high level. Among them are Schorr's Register Transfer Language [11], Schlaeppi's LOTIS language [10], a FORTRAN dialect by Metze and Seshu [8], and ALGOL dialects by Gorman and Anderson [6], and by Chu [2]. Each of these languages

Manuscript received October 4, 1968.
The authors are with the IBM Watson Research Center, Yorktown Heights, N. Y.

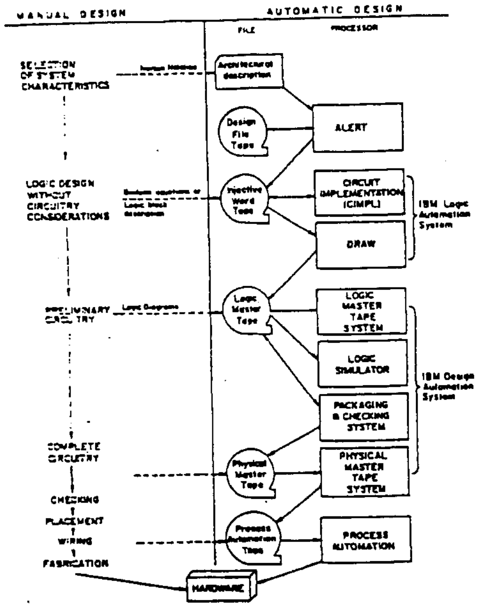


Fig. 1. Successive stages of development of a new computer and associated automation systems at IBM.

provides a means to formalize the information needed by architectural, engineering, design, and programming groups during the design of a computer. These languages convey differing degrees of detail in the representation of a machine, although each provides considerable flexibility in the amount of detail that may be included. In the case of the ALERT project, a descriptive language was chosen which is based on Iverson notation [7] which offers exceptional flexibility, convenience, and conciseness without sacrificing desirable features of the languages previously mentioned. Iverson notation has been applied in several fields, especially in the APL interpreter [3], and it has been used extensively as a high-level computer description language [4]. In particular, Iverson notation is an algorithmic language, which makes it possible to represent complex sequential logic as programs of microevents.

Some familiarity with the conventions of Iverson notation is desirable for proper understanding of the discussion in this report. The reader is referred to the summary on pp. 196–203 of [4].

It was necessary to develop additional conventions to depict system structure and data flow information for this application. Although Iverson notation has previously been used to describe the programming and functional characteristics of computer systems, these programming descriptions have not depicted the particular mechanisms used in those systems. To apply Iverson notation as a design specification language, the most important new convention imposed was that variables in the descriptions are considered to represent physical devices such as flip-flops, gate output lines, or registers. Because such devices constitute permanent hardware, declaration statements are needed to fix categories and dimensions of variables. A variable is automatically recorded as a single bit at its first occurrence in a description unless declared otherwise.

In accordance with this constraint, several Iverson operators which serve to modify operand dimensions or rank, such as compression or expansion, are excluded.

The entire computer description is separated into microprograms which describe data transformations in the computer. Each microprogram is considered to possess individual timing and control logic so that it may execute concurrently with other microprograms.

In addition to the regular microprograms, special macro functions may be defined in computer descriptions. Such functions can be invoked by and included as subsections in other microprograms.

Declaration statements are distinguished by a $D$ in column one of the punch card, the start of microprograms by an $M$ (or $MM$ if manual specification of control logic is desired), and macro functions are denoted by an $F$. In addition, new statement types were introduced, such as assignment statements which affect only set or reset lines of flip-flops. Optional symbolic statement labeling is permitted.

Because the input is punched on cards to be accepted by the 7094, symbols in Iverson notation must be transliterated as shown in Table I. An innovation in this version of Iverson notation is the period used as statement terminator: all entries to the right of a period in a card are considered to be comments.

A complete example of a computer description for ALERT processing may be found in [5].

## Mechanization of the Logic Design Process

At this point, specific techniques will be considered for processing the input description to obtain an acceptable logic design.

Design by humans involves strategic planning and subsequent development to carry out the plans. The goal in the ALERT project was to delegate development work to machines whenever possible. This requires that strategic aspects of design must be distinguished from development.

Unfortunately, there is not always a clear distinction between these two aspects of design. For example, the choice of a computer instruction repertoire is clearly strategic and must involve the human. Connections of logical components in a binary tree decoder is generally more routine and would be appropriate to delegate to machines. But certain details of the control logic, for example, may be best determined by human judgment while other details fall into place in a mechanical way.

DEF011970

TABLE I
TRANSLITERATION OF APL SYMBOLS FOR KEYPUNCHING MACHINE DESCRIPTIONS FOR ALERT

| Symbol | Meaning | Transliteration |
|---|---|---|
| $+ \; - \; \times \; \div$ | Arithmetic operations | + − * 'DIV' |
| * | Exponentiation | ** |
| $\wedge \; \vee \; \sim$ | Logic operations | 'AND' 'OR' 'NOT' |
| \| | Residue, absolute value | 'RESIDUE' 'ABS' |
| $< \; \leq \; > \; \geq$ | Relations | 'LS' 'LE' 'GR' 'GE' |
| $= \; \neq$ | | 'EQ' 'NE' |
| $\alpha$ | Prefix | 'PREFIX' |
| $\omega$ | Suffix | 'SUFFIX' |
| , | Catenation | , |
| / | Reduction | / |
| $\perp$ | Base 2 Value | 'VALUE' |
| $\phi$ | Left, right shift | 'LEFT' 'RIGHT' |
| $\leftarrow$ | Assignment | = |
| $\rightarrow$ | Branch | GO TO |
| $\rightarrow(X)/S$ | Conditional branch | IF (X) GO TO S. |
| [ ] | Subscript | ( ) |
| ; | Subscript delimiter | ., |
| : | Label | .. |
| Carriage return | End of statement | . |

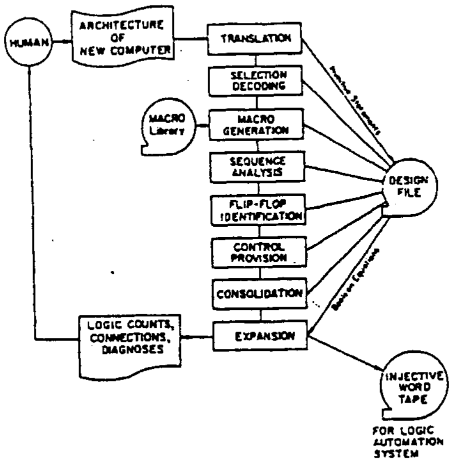


Fig. 2. Transformation steps of ALERT.

It is not desirable to impose a priori rules concerning what the user should and should not specify. Here the flexibility of Iverson notation proves valuable. Default options are provided so that the designer can work at the level of detail he desires. He may choose to omit details, and in those cases the system will refer to the default option to provide the details according to predefined conventions.

To mechanize the design process, a series of explicit transformation steps have been defined, each of which serves to bring the computer description closer to the final logic design. Each step consists of a set of subroutines concerned with some aspect of the design process. The steps systematically transform the machine description by supplying detail needed to complete the logic design.

The transformation steps are provided only as a convenience to the user so that he may avoid detailed work. Whenever the user wishes to specify details explicitly, he may override the system.

There is considerable latitude possible in the choice and order of these steps. In ALERT, the design process was separated into eight major steps called translation, selection decoding, macro generation, sequence analysis, flip-flop identification, control provision, consolidation, and expansion. (See Fig. 2.)

During the successive transformations from architecture to logic, the desired computer is represented internally by a single common format. The internal representation language is sufficiently general to depict a computer at the high architectural levels or down to a bit-by-bit account of the logic. The representation resides in the master design file, a threaded linked-list structure organized for rapid retrieval and manipulation of data.

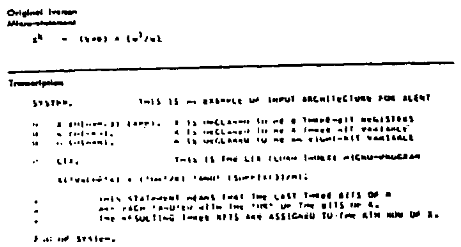
To apply the ALERT system, just as in the traditional approach the user first selects overall features of the desired machine, and then he must formulate a machine organization to provide those features. The instruction set, major registers, and data flow paths should be identified, but routine details involving timing control, addressing logic, or hidden registers may be passed over and left for ALERT to work out. At this point the user is ready to prepare the architectural description.

The architectural description begins with declarations of the storage, channels, important registers, toggles, and other devices which the user wishes to note. Predesigned blocks of logic such as decoders or adders may be described as macro functions. Data flow paths are represented as register transfers using assignment statements. Memory access, indexing, and interrupt processing may be specified as microprograms. The instruction set can be indicated as a set of microprograms, and the overall control should be specified as a master microprogram. Such a document serves as a formal high-level description of the intended machine.

The user punches the architectural description onto cards and inputs it to the processor. The eight steps of the ALERT system then successively transform that description into logic.

In the following section, we shall examine the transformation steps in terms of their purposes and methods. To exemplify the processes, we shall show how each step helps to transform a typical Iverson statement from architecture to logic.

The sample statement taken from an architectural description written by A. Falkoff defines a microprogram algorithm to accomplish the load index instruction



```
Original Iverson
Micro-statement
    X^k  ← (k≠0) ∧ (ω³/m)

Transcription
    SYSTEM.              THIS IS AN EXAMPLE OF INPUT ARCHITECTURE FOR ALERT

    X [illegible] ARRAY.   X IS DECLARED TO BE 8 THREE-BIT REGISTERS
    K [illegible].         K IS DECLARED TO BE A THREE BIT VARIABLE
    M [illegible].         M IS DECLARED TO BE AN EIGHT-BIT VARIABLE

    LIX.                 THIS IS THE LIX (LOAD INDEX) MICRO-PROGRAM

        [illegible]

            THIS STATEMENT MEANS THAT THE LAST THREE BITS OF M
            ARE EACH [illegible] WITH THE [illegible] OF THE BITS OF K.
            THE RESULTING THREE BITS ARE ASSIGNED TO THE KTH ROW OF X.

    END OF SYSTEM.
```

Fig. 3. A microstatement in Iverson notation and its transcription keypunched as a computer description.

in a small computer. It was originally specified in Iverson notation as

$$X^k \leftarrow (k \neq 0) \wedge (\omega^3/m).$$

In this statement, $X$ represents a matrix of eight 3-bit index registers, $m$ is an 8-bit register, and $k$ is an integer variable between 0 and 7.

This statement indicates selection of row $k$ of $X$, which in fact identifies index register number $k$. That register is loaded with the value of the last three bits of $m$, provided that $k$ is not equal to zero. If $k$ is zero, then the number $k$ index register (i.e., index register number 0) is set to zero.

This statement appears deceptively simple, and thereby illustrates the conciseness of Iverson notation. It implies existence of gating to set or reset any of the eight 3-bit index register, i.e., 24 bits. It also implies existence of a mechanism to select the appropriate register as well as provisions to avoid data transfer when $k=0$.

To transcribe this statement, $k$ was changed from a scalar integer variable to a bit string (which will in fact be three bits long). For keypunching we represented this bit string as "$K$." The expression "$k \neq 0$" then was reformulated as "$(\perp K) \neq 0$," or more simply, as "$\vee/K$."

In order to elucidate the nature of the ALERT transformation steps, this statement was processed in isolation as though it were an entire machine description. A complete input deck was prepared describing an imaginary computer which does nothing except load its $k$th index register. This deck is listed in Fig. 3 with the names and Iverson operators transliterated for keypunching along with the required declaration and heading statements.

We will now examine the individual ALERT steps and their effects upon this machine description.

## The Transformation Steps of ALERT

The first routine to process the input was the translator.

### *Translation*

This routine scans the modified Iverson notation and converts it into a more constrained format used in the design file. This routine also checks the input for format, syntax, and consistency. The input language permits considerable syntactic complexity including indefinitely nested expressions, symbolic statement labeling (labels may be referenced with variably subscripted names), and context determined interpretation of operations (as described below in the section "Constants in a Description").

The internal design file representation is more restricted. The file is composed of fixed-format "primitive" statements, each of which may depict only a single type of gating function along with the signal sources and destinations. These primitives provide a framework upon which the desired logic design is built. The translation then is a parsing, analysis, and decomposition of the input statements into a simpler more design-oriented form. The procedure-oriented input reflects data flow and control transactions, while the design file more directly indicates componentry and paths implied by that flow. During translation, intermediate variables implied by Iverson expressions but not explicitly represented in the microprograms are generated and filed along with the declared elements.

Translation is a literal reformulation of the architecture, and the user must take care to express his input so as not to imply an obviously redundant structure. This need occurs in the example in Fig. 3. The expression "$(k \neq 0)$" in the original Iverson statement could be rewritten as "$((\perp K) \neq 0)$" when keypunched, but this would imply existence of a complicated mechanism to compare the value of the bits of $K$ with a special vector whose bits have the value zero. The expression "$(\vee/K)$" is preferable because it implies only a simple OR gating which achieves the same effect as the more elaborate mechanism.

The translator also must assemble the statements into the proper design file encoding. As each variable name is scanned, it is relegated to a name list. The name's address in this list is not directly entered into the design file, but a threading is calculated for all references to that variable.

The design file is forward-threaded so that each reference to a variable name is actually a pointer to the address of the next occurrence of that name in the file. Only the last occurrence identifies the variable itself. Threading makes it possible to retrieve all references to any variable with a minimum number of file accesses. The entries of the design file are packed two to a 36-bit 7094 word. Four bits are used to designate type of entry and 14 bits constitute the entry itself. The purpose of packing is to double the number of entries in the file (but, of course, at the expense of additional processing needed to pack and unpack entries).

Experience using ALERT has indicated that threading successfully saves processing time, but that the data packing is of less critical value, especially since the design file must overflow onto tape for a machine design of substantial size.

```
LABEL=        L1
 INPUT=       K
   GATING=    OR
    OUTPUT=   T2
 INPUT=       T2
 INPUT=       M        0003 BITS. OFFSET=0005
   GATING=    AND
                       0001 BIT.  OFFSET=K
    OUTPUT=   X        0003 BITS. OFFSET=0000
```

Fig. 4. Printout of the design file after the translation step.

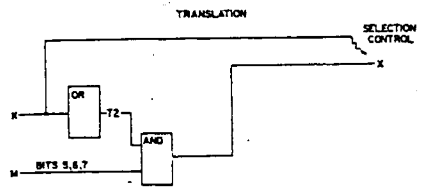


Fig. 5. Schematic diagram of the structure represented in the file after translation.

Fifteen types of entries are distinguished, including signal sources and destinations, subscripts (constants or variables), statement labels, microprogram headings, gating operators, patching markers, and even pointers to comments (comments may be recorded on a separate tape).

It may be instructive to examine the design file printout in Fig. 4 to observe the result of translating the sample machine description presented in Fig. 3.

Two design file statements were produced signifying two gating structures. The first statement (lines 2, 3, and 4 of the listing) indicates that the (three) components of register $K$ are connected to a single OR gate. To represent the output of this gate, a new variable name, T2, was generated by the translator. This variable, which represents a single wire, is connected to a structure of AND gates according to the second design file statement. These gates serve to inhibit or pass three consecutive bits from $M$ beginning with a bit five positions from the left end which corresponds to the expression "(SUFFIX(3)/$M$)." The outputs from these AND gates are specified as having for their destination all three bits of the $K$th row of the array $X$.

The reader should not be discouraged by the intricacy of this explanation since the format is not organized for human reading. Details of the design file encoding such as identification of array components (the SIZE and OFFSET in Fig. 4) are properly beyond the scope of this report. In the following sections we will instead refer primarily to diagrams that were drawn by hand which depict the structures represented by the contents of the design file. Fig. 5 is such a diagram showing the structure represented in the file after translation. The reader may note a one-to-one correspondence of file entries and elements of the diagram. Note that, in this and successive diagrams, individual elements are not necessarily distinguished in arrays.

After the entire architectural description has been processed by the translation routines, another group of routines further transforms the description in the design file. These constitute the selection decoding step.

*Selection Decoding*

It is sometimes useful in a computer description to attach a variable subscript to an array name, as for instance in the expression $X$('VALUE' $K$) in Fig. 3. This is taken to mean that component elements of the array ($X$ in this case) may be selectively addressed during operation of the proposed machine, and that they are addressed according to the value of the subscripting variable ($K$ in this case). Variable subscripting provides a shorthand for representing complicated logic structures.

In ordinary program-generating compilers such as FORTRAN, variables represent values rather than structures, and variable subscripts are readily processed by generating code to add the value of the variable to the array address during execution. In a design compiler, variable subscripts present more difficulty. Selection of an element in an array requires sufficient logic to open the data paths to the desired element, but to no others, according to the contents of the selecting variable. Moreover, the selection logic for arrays used as signal destinations is quite different from that for arrays used as signal sources.

The selection decoding programs scan the design file and when a variable subscript is discovered, it is replaced by a block of logic to provide selection. In particular, the output of the subscript variable is fed to a decoder to produce a selection vector. Each bit of this selection vector is gated to control availability of one element of the array. Arrays of one, two, or three dimensions are permitted and up to three variable subscripts may be used to reference an array.

In the case of the sample design, the variable subscript in the term $X$('VALUE' $K$) signifies that components of the array $X$ are selected according to the current value of the bits in $K$. Since this expression occurs on the left side of an assignment statement, the selection decoding routines recognize that the selection affects the input lines to $X$. This capability is provided by supplying a decoder tree for the bits from $K$. The output from this tree is used to control access to the $X$ bits. Fig. 6 lists the contents of the design file after these automatic modifications have been made, and Fig. 7 is the manually prepared diagram of the design at this stage. The decoder is represented by a macro operator which is processed by the next step.

*Macro Generation*

In addition to the elementary logic operators, various higher order functions may be specified in the machine description. In the preceding transformations, such high-order operators were treated as black boxes. The

```
LABEL=         L13
 INPUT=        K
   GATING=     OR
    OUTPUT=    T2
  INPUT=       T2
  INPUT=   .   M          0003 BITS, OFFSET=0005
    GATING=    AND
     OUTPUT=   T3
  INPUT=       K
    GATING=    DECODE
     OUTPUT=   T4
  INPUT=       T4
  INPUT=       T3
    GATING=    AND
    COUPLING=  CC12
   ·OUTPUT=    X
```

Fig. 6. Printout of the design file after selection decoding.

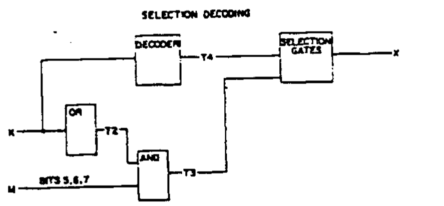


Fig. 7. Schematic diagram of the structure after selection decoding.

macro generator replaces these black boxes by the complete combinational logic required to accomplish those operations. This involves replacing any occurrence of a high-order operator in a microstatement by an entire block of elementary logic. Such a process is similar to generation of detailed coding by an ordinary assembler program when a macro is encountered, and the techniques used are analogous to those of conventional macro processors.

These blocks of logic may be copied literally from libraries of such blocks, or they may be custom-generated by logic-designing algorithms. In addition to the high-order functions specified by the designer in the original architecture, macros are supplied for decoders, counters, and shifters.

Decoders, for instance, are generated as simple binary trees. Other high-order operations, especially the arithmetic functions, may entail a number of special considerations. There are many alternative designs, for example, of adders. If the user does not specify his own macros, default macros are automatically provided.

In the sample design, a macro reference to a decoder was entered into the file during the previous step. During the current step, the design file was scanned by the macro generator and the operator "DECODE" was recognized as a macro name. Logic for a binary decoding tree was generated using the arguments $K$ and T4 as the decoder's input and output, respectively. The routine automatically determined the size of the decoding tree

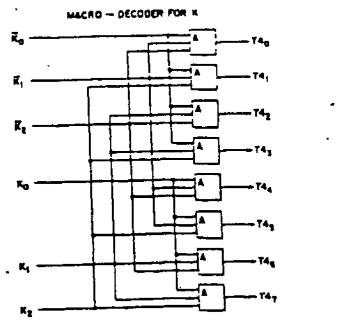


Fig. 8. Diagram of the decoder produced by the macro generator.

according to the number of bits of the arguments. The design produced by the macro processing is depicted in Fig. 8.

### *Sequence Analysis*

During this step, control and sequencing requirements for the design are determined.

The user may choose to specify explicitly all control logic within sections of the design where timing or other constraints warrant particular attention to detail (as in memory accessing, for instance). Within less critical sections of the design, however, the user may omit routine details of the control. This approach is called *implicit* timing, and control is automatically derived according to the sequence inherent in the microprogram algorithms of the architectural description.

An algorithm specifies a discrete sequence of microevents. But it is not desirable to assign every microevent a distinct time period in a sequential network. Instead, the microevents are grouped together into as few separate time periods as is possible. This will allow concurrent parallel operations.

The sequence analysis routine partitions the microprogram into groups of operations which may occur during the same time period or event. In the case of those microprograms having manually specified control logic, this processing is not required. In the case of microprograms with implicit timing, the sequence analysis program must distinguish successive control periods. This is accomplished by scanning each microprogram in the design file from its beginning and determining points where new event groups must occur.

A microprogram is partitioned into event groups by the following algorithm.

1) The beginning of each microprogram is established as the partition point of an event group.

2) Statements which are destinations of GO TO's are established as partition points. (GO TO's signify deviations from sequential flow of microprocesses.)

3) Conditional test statements (IF's) are followed by partition points.

4) Each event group is scanned from its initial partition point, and whenever any variable previously encountered in that group is to receive a signal, then the statement under examination is established as a new partition point.

The partition points are recorded in a sequence of events table which is later used to provide control logic.

ALERT was programmed so that each implicit microprogram is provided with a single series of event table entries. It should be mentioned that as an alternative approach, a microprogram could be separated into subsections, each having distinct sequences. In this case, the total number of events could be reduced, but additional control would be needed to distinguish the subsections.

In the table, the event numbers may not follow a strictly one-by-one progression. Certain gating operations may continue for a period of several time units when they are performed, and the table entries must reflect this. Furthermore, the designer may set the next event count to a value relative to a previous statement in order to impose additional timing constraints.

When the sample design previously mentioned was processed, the sequence analysis determined that all microprocesses could be accomplished within a single time period. Just one partition point was established at the beginning of the microprogram. Two event periods, however, are required. The first period (event number 0) is an inactive phase required for every microprogram to indicate that it is not in execution. The other period (event number 1) is the single active phase.

### *Identification of Flip-Flops*

In the previous steps, we did not distinguish between logic structures which produce output only momentarily while they receive input, and structures which must attain states which persist after their input ceases. The purpose of this routine is to identify variables which need to retain their values over some period of time.

The microprograms are scanned for any variable which serves as a signal source during a later event period than when it acquired its value. Such a variable must be able to hold its value, and accordingly it is identified as a flip-flop. A flip-flop (or latch) is a bistable circuit element which stores a bit of data until some new input alters its state. The value of this datum is available as the flip-flop's output.

In addition to this automatic search for flip-flops, the user may declare variables to be flip-flops by writing "(=FF)" following the variable's name (as in the case of $X$ in Fig. 3). To avoid possible omissions, the user should declare any variable to be a flip-flop when he intends it to be so. Likewise, whenever he intends a name to represent a register, he should declare it to be a vector of flip-flops.

Every flip-flop is considered to have a SET input which puts the device into the "ONE" state, and a RESET input which puts it into the "ZERO" state. Two new types of assignment operators were provided to correspond to these inputs, namely "='SET'" and "='RESET'". A statement of the form,

$$A = \text{'SET'}\ B,$$

specifies that the output from bit $B$ is connected to the SET input line of flip-flop $A$. Likewise,

$$A = \text{'RESET'}\ B,$$

specifies that the output from $B$ is connected to the RESET input of $A$. For formal completeness, we may define these assignment operators as:

$$A = \text{'SET'}\ B \quad \text{means } A \leftarrow A \vee B;$$

$$A = \text{'RESET'}\ B \quad \text{means } A \leftarrow A \wedge \overline{B}.$$

A 0 or 1 signal delivered to the SET input of a flip-flop causes the device to acquire its previous state ORed with the value of the SET line. RESET input of value 0 causes the flip-flop to remain at its previous value, while a RESET input of value 1 assigns the value 0 to the flip-flop.

If a variable $P$ in the design file is discovered to be a flip-flop, then any design file statement of the form

$$P = Q$$

will automatically be rewritten as two statements:

$$P = \text{'SET'}\ Q,$$
$$P = \text{'RESET'}\ \text{'NOT'}\ Q.$$

This specifies that $Q$ and its complement $\overline{Q}$ are connected to the SET and RESET inputs of $P$, respectively. These two connections serve to jam the value of $Q$ into flip-flop $P$.

Control signals for passing or blocking data to flip-flops must be distinguished from the data themselves. The control signals, like the data, must be duplicated and connected to the SET and RESET sides of flip-flops, but unlike data, control must not be complemented at the RESET side. This allows precisely the same flip-flops to be selected for SET and for RESET operations.

Referring to the sample design as developed in Fig. 7, the output lines from the selection gates are loaded into $X$. But $X$ was declared to be a flip-flop array. The flip-flop identification step modifies this logic so that the positive values of these lines are directed to the SET input terminals of the $X$ flip-flops. However, a second bank of gating is then automatically provided to supply complemented input data to the RESET terminals of $X$. (See Fig. 9.) Note, however, that NOT gates are used only to complement the data (the T3 lines), but that the selection vector, T4, which controls access to the appropriate components of $X$, is not complemented.

Content paper 3 page(s) 600 & 601 is missing from the original File History.

Please pardon any inconvenience.

1) This routine searches for identical operators which have the same inputs. Such operators are respecified as one consolidated operator, and the original event control signals are used to enable each output at its appropriate time.

The original inputs and outputs are gated by their respective event control signals.

For example, suppose the following statements occur in the design file.

Statement no. [1]: $A = B$ 'AND' $C$.
[2]: $D = E$ 'OR' $A$.
[3]: $F = D$ 'AND' S1(1).
⋮
[24]: $G = B$ 'AND' $C$.
[25]: $H = E$ 'OR' $G$.
[26]: $J = H$ 'AND' S1(4).

Here S1(1) and S1(4) are timing control signals. The consolidation routine eliminates statement [24] and then $G$, wherever it occurs, is replaced by $A$. Statement [25] then becomes $H = E$ 'OR' $A$. This new statement is, in turn, eliminated and $H$ is replaced by $D$. Statement [26] will finally become $J = D$ 'AND' S1(4).

This process is repeated until no identical gates are found in the file.

2) Another function of consolidation routine is to reduce gatings by rearranging the logic.

Until consolidation, arrays have been depicted as simple monolithic entities. Connections involving vectors or matrices were represented as though only single units were joined when in fact arrays of connections and gates were implied. Many more lines and gates may be required for the connections that were actually chosen than would be needed for logically equivalent connections.

The routine examines logically equivalent variations of the expressions and counts the components needed for each variation. The variation requiring fewest elements is selected.

3) If signals are assigned to a flip-flop in more than one micro-statement, the signal sources are ORed together so that only one assignment is provided. In this way, all signals used to SET or RESET any flip-flop are collected into common connections.

The result of consolidation on the sample design is shown in Fig. 11. Because of its small size, only the rearrangement part (item 2) of the consolidation step affects this design. The consolidation routine discovered that the gating arrangement in Fig. 10 is wasteful. The S1(1) signal is reconnected so that it is ANDed with T4 instead of T14 and T16 as in Fig. 10. The number of individual gates was reduced by this process from 96 to 56.

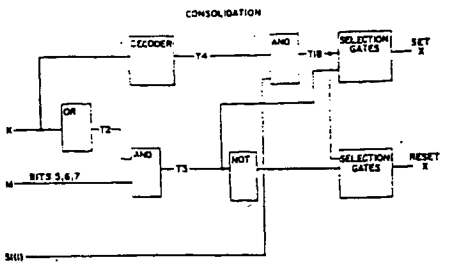


Fig. 11. Diagram of the structure after consolidation.

## *Expansion*

By this point the logic design is nearly complete, and it only remains to output the results in the form of Boolean equations.

The design file entries, however, may actually represent arrays of connections. A single statement may signify a large number of individual connections, and these may be connected in complicated patterns.

The purpose of the expansion step is to report every device and connection individually. During this step, each statement in the design file is examined, and when variables are discovered to comprise arrays rather than single bits, separate copies of the original statements are generated for the separate components of the array. As the statements are recopied, component identifications are calculated according to control terms in the file. The expansion results in such a marked enlargement of the file that the threading conventions become inconvenient and are dropped at this point. An exhaustive connection-by-connection exposition of the logic is produced.

When the sample design was expanded, $K$ was found to be a three-bit vector, $M$ involved three bits, and 24 SET lines and 24 RESET lines were required for the eight three-bit index registers denoted by $X$. In addition, the intermediate system-generated variables T3, T4, and T18 were discovered to be arrays. Therefore, statements which involved these variables were repeatedly copied with components individually identified which resulted in 140 gating statements. Seventy-two of these statements represent "gates" in only a purely formal sense since they are actually a renaming of an element. (The renaming is due to techniques used in processing. It is not of concern since these "gates" are automatically eliminated later.)

A printout of the design at this point in Boolean equation format is shown in Appendix I. The first equation in this listing,

$$T2 = K/0 + K/1 + K/2,$$

signifies that the three bits of $K$, i.e., $K(0)$, $K(1)$, and

$K(2)$,[1] are connected to a common OR gate, the output of which is denoted T2. The next three statements indicate how this output is connected with the three bits from $M$ to produce a three-bit variable denoted as T3. The next equations present an example of a "formal" intermediate variable and a "formal" gating function. This is actually only a renaming of a variable for bookkeeping purposes: the variable T5 is exactly the same as the variable $K$. The remainder of the printout may be interpreted similarly.

At this point, processing by the ALERT programs has ended. As mentioned earlier, other automation systems transform logic equations into circuitry, and these are now called in to complete the machine design which has been developed by the ALERT program.

## Further Processing After ALERT

The IBM logic automation system accepts input in the form of Boolean equations as produced by the ALERT system. This system accomplishes a type of Boolean minimization and then converts the resulting logic into circuit gates of the IBM solid logic technology family. This entails reorganization of the logic to satisfy circuit engineering requirements. The logic automation system also lays out the resulting logic in engineering diagrams. Finally, the design is recorded into standard engineering files by the IBM design automation system and from these files the machine may be simulated, tested, and manufactured.

When the sample design in Appendix I was processed by the logic automation programs, it was discovered that 72 "gates" were actually only renamings of variables (as noted in the preceding section), and therefore these "gates" were eliminated. The program also discovered that the variable T2 is precisely the complement of another variable, T4(0), and accordingly T2 was eliminated. After the logic automation minimization step, 67 logic gates remained. These, however, are ideal logic gates which do not conform to the engineering constraints of solid logic technology circuitry. Only a limited choice of circuits in this technology were available to provide gating functions, and these did not match the ideal gates specified. The logic automation program automatically replaced the ideal gates in the design with available circuits and then proceeded to supply additional gates to correct for factors such as module input pin restrictions, fan-out limitations, and signal polarity inversions caused by the circuits. When all modifications were made, 191 gates resulted. These were automatically diagrammed by the logic automation system into nine pages of logic sheets, as shown in Appendix II.

In the first diagram, for example, lines from the bits of $K$ are passed through inverters (denoted by $N$) and connected to an AND-inverter gate labeled "3C-CC." Notice a line drawn off this gate's output labeled "T4 0." This is the complement of the variable T4(0) which is one of the decoded signals formed from the bits of $K$, and it corresponds to the variable T2.

Although there has been considerable rearrangement of the logic during the course of the design, the reader may trace out in the nine diagrams all functions originally specified by the Iverson notation. The diagrams constitute an "implementation" of the architectural description in solid logic technology circuits. The circuitry was generated directly from the architecture untouched by human hands.

## Conclusion

The sample architecture description in Fig. 3 was chosen to elucidate the operation of the ALERT system. Actual computer descriptions are, of course, far more extensive. However, the example illustrates the large quantity of detail that may be generated from a simple statement in Iverson notation. The machine description in the exact form shown in Fig. 3 was processed and the actual results of each processing step are depicted in Figs. 4 through 11 and Appendix I. Appendix II presents machine-drawn diagrams which were automatically derived from the final design produced.

ALERT thus is an operational logic generator. Several experimental techniques were introduced to assist logic design. Some techniques, such as selection decoding, appear to aid the designer only occasionally, while other processes, such as macro generation, seem widely useful and warrant further development.

The results suggest that a compiler of logic may provide the sort of improved usage of design manpower that programming compilers have provided to programmers. This implies faster development of new computers, immediate documentation, and also improvement of quality because the architect/designer may try alternative plans without waiting for manual logic design and choose the best from among these preliminary designs.

## Appendix I

### Printout of Boolean Equations Generated by the Expansion Step

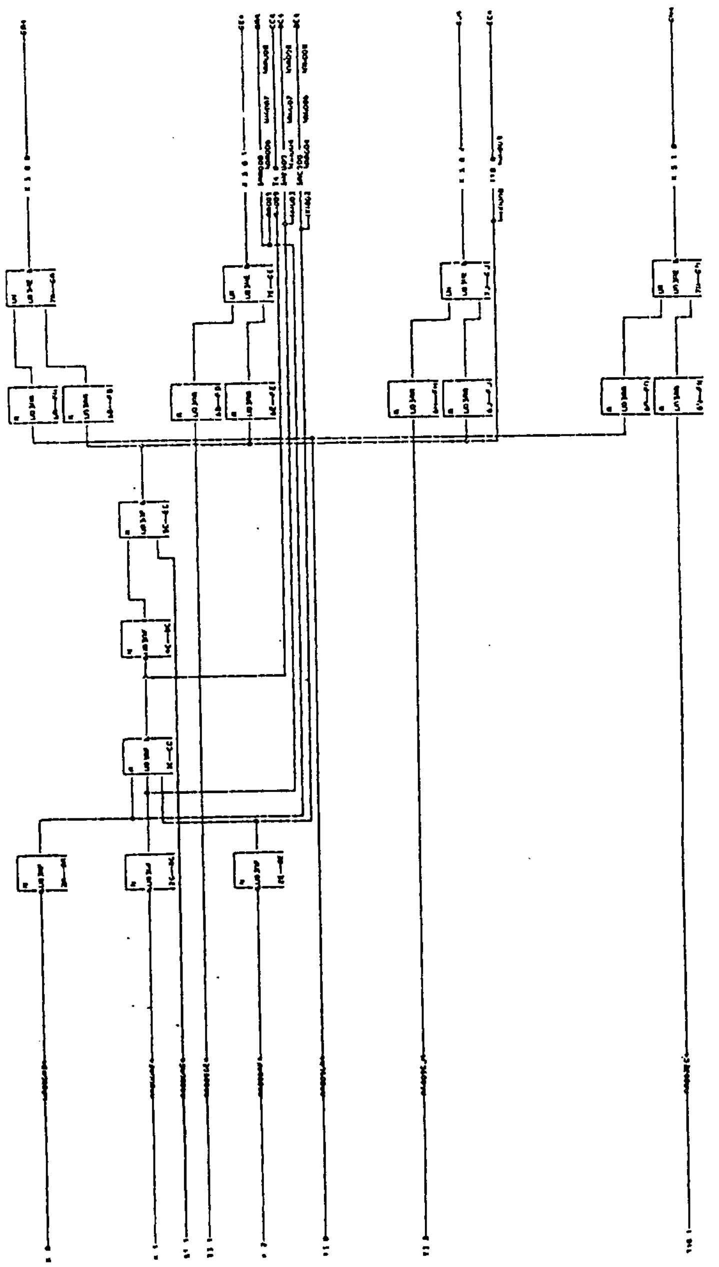
```
T2       =    K/0+ K/1+ K/2
T3/0     =    T2* M/5
T3/1     =    T2* M/6
T3/2     =    T2* M/7
T5/0     =    K/0
T5/1     =    K/1
T5/2     =    K/2
T4/7     =    T5/0* T5/1* T5/2
```

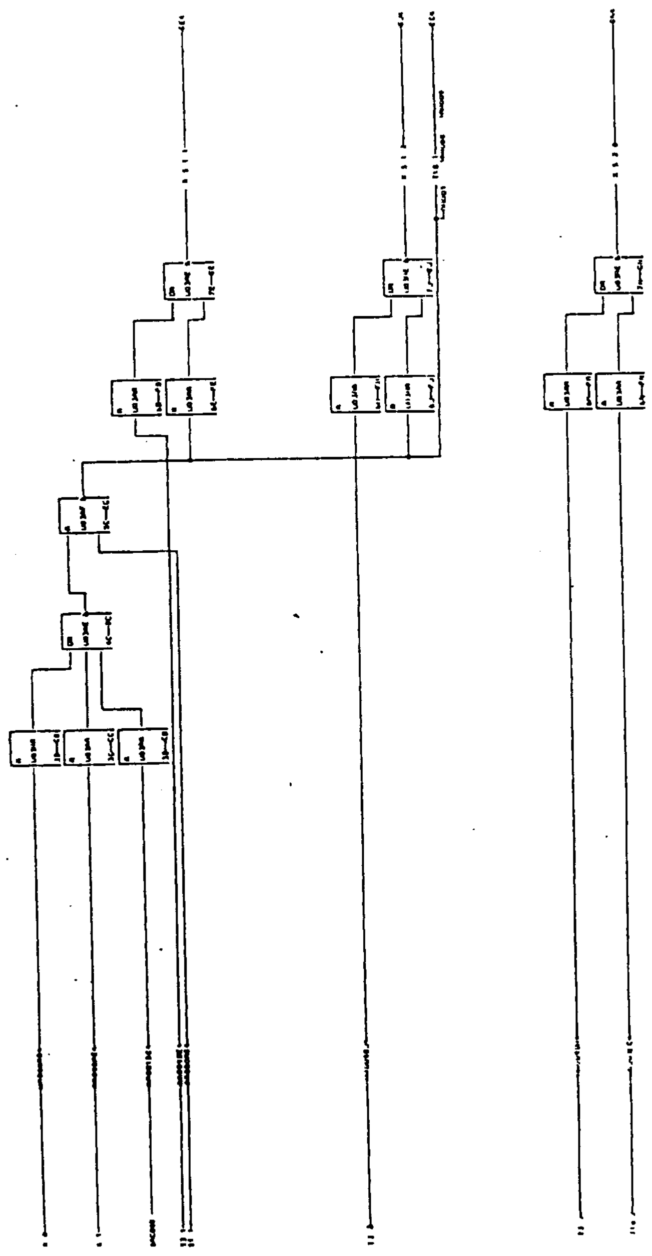
[1] Unfortunately, formatting conventions for arrays differ in ALERT, in the Boolean equation format, and in the engineering diagrams since the three systems were developed by separate groups. The variable "$K(0)$" in ALERT is represented as "$K/0$" in the logic equation format, and this same variable is printed out as "$K0$" in the engineering diagrams. Likewise, NOT in logic equation format becomes a minus sign, OR a plus sign, and AND an asterisk.

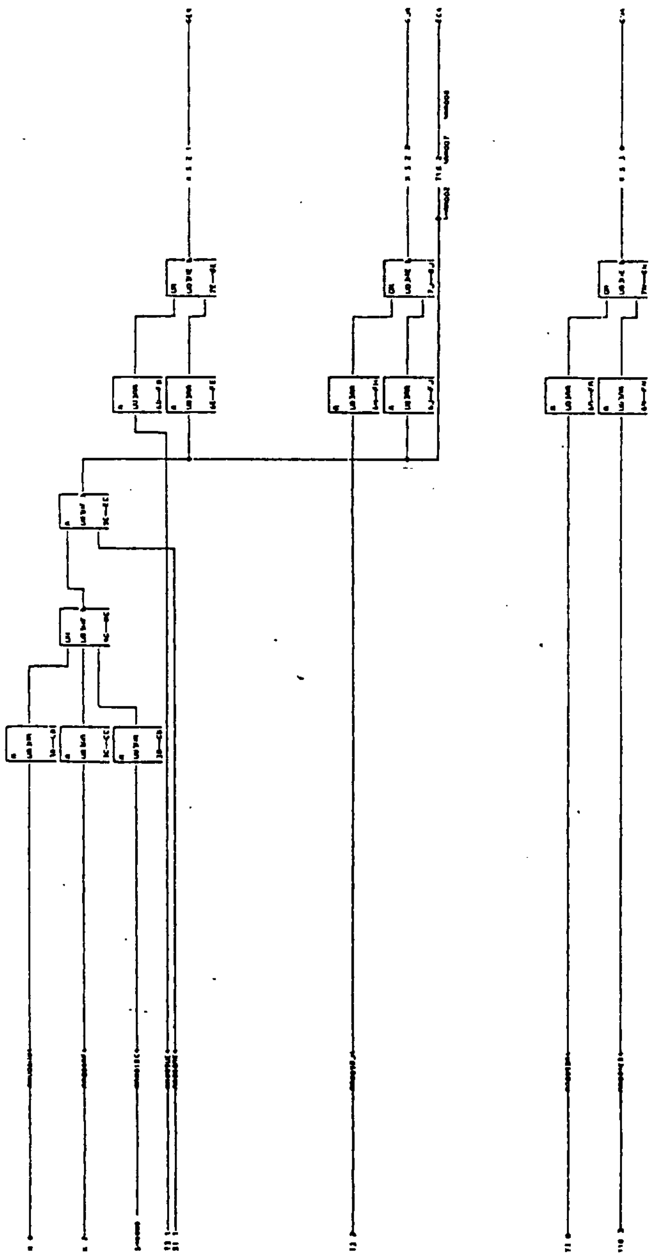
```
T6/0       =   K/0
T6/1       =   K/1
T6/2       =   -K/2
T4/6       =   T6/0* T6/1* T6/2
T7/0       =   K/0
T7/1       =   -K/1
T7/2       =   K/2
T4/5       =   T7/0* T7/1* T7/2
T8/0       =   K/0
T8/1       =   -K/1
T8/2       =   -K/2
T4/4       =   T8/0* T8/1* T8/2
T9/0       =   -K/0
T9/1       =   K/1
T9/2       =   K/2
T4/3       =   T9/0* T9/1* T9/2
T10/0      =   -K/0
T10/1      =   K/1
T10/2      =   -K/2
T4/2       =   T10/0* T10/1* T10/2
T11/0      =   -K/0
T11/1      =   -K/1
T11/2      =   K/2
T4/1       =   T11/0* T11/1* T11/2
T12/0      =   -K/0
T12/1      =   -K/1
T12/2      =   -K/2
T4/0       =   T12/0* T12/1* T12/2
T18/0      =   T4/0* S1/1
T18/1      =   T4/1* S1/1
T18/2      =   T4/2* S1/1
T18/3      =   T4/3* S1/1
T18/4      =   T4/4* S1/1
T18/5      =   T4/5* S1/1
T18/6      =   T4/6* S1/1
T18/7      =   T4/7* S1/1
T15/0/0    =   T18/0* T3/0
T15/0/1    =   T18/0* T3/1
T15/0/2    =   T18/0* T3/2
T15/1/0    =   T18/1* T3/0
T15/1/1    =   T18/1* T3/1
T15/1/2    =   T18/1* T3/2
T15/2/0    =   T18/2* T3/0
T15/2/1    =   T18/2* T3/1
T15/2/2    =   T18/2* T3/2
T15/3/0    =   T18/3* T3/0
T15/3/1    =   T18/3* T3/1
T15/3/2    =   T18/3* T3/2
T15/4/0    =   T18/4* T3/0
T15/4/1    =   T18/4* T3/1
T15/4/2    =   T18/4* T3/2
T15/5/0    =   T18/5* T3/0
T15/5/1    =   T18/5* T3/1
T15/5/2    =   T18/5* T3/2
T15/6/0    =   T18/6* T3/0
T15/6/1    =   T18/6* T3/1
T15/6/2    =   T18/6* T3/2
T15/7/0    =   T18/7* T3/0
T15/7/1    =   T18/7* T3/1
T15/7/2    =   T18/7* T3/2
X/S/0/0    =   T15/0/0
X/S/0/1    =   T15/0/1
X/S/0/2    =   T15/0/2
X/S/1/0    =   T15/1/0
X/S/1/1    =   T15/1/1
X/S/1/2    =   T15/1/2
X/S/2/0    =   T15/2/0
X/S/2/1    =   T15/2/1
X/S/2/2    =   T15/2/2
X/S/3/0    =   T15/3/0
X/S/3/1    =   T15/3/1
X/S/3/2    =   T15/3/2
X/S/4/0    =   T15/4/0
X/S/4/1    =   T15/4/1
X/S/4/2    =   T15/4/2
X/S/5/0    =   T15/5/0
X/S/5/1    =   T15/5/1
X/S/5/2    =   T15/5/2
X/S/6/0    =   T15/6/0
X/S/6/1    =   T15/6/1
X/S/6/2    =   T15/6/2
X/S/7/0    =   T15/7/0
X/S/7/1    =   T15/7/1
X/S/7/2    =   T15/7/2
T17/0/0    =   T18/0* -T3/0
T17/0/1    =   T18/0* -T3/1
T17/0/2    =   T18/0* -T3/2
T17/1/0    =   T18/1* -T3/0
T17/1/1    =   T18/1* -T3/1
T17/1/2    =   T18/1* -T3/2
T17/2/0    =   T18/2* -T3/0
T17/2/1    =   T18/2* -T3/1
T17/2/2    =   T18/2* -T3/2
T17/3/0    =   T18/3* -T3/0
T17/3/1    =   T18/3* -T3/1
T17/3/2    =   T18/3* -T3/2
T17/4/0    =   T18/4* -T3/0
T17/4/1    =   T18/4* -T3/1
T17/4/2    =   T18/4* -T3/2
T17/5/0    =   T18/5* -T3/0
T17/5/1    =   T18/5* -T3/1
T17/5/2    =   T18/5* -T3/2
T17/6/0    =   T18/6* -T3/0
T17/6/1    =   T18/6* -T3/1
T17/6/2    =   T18/6* -T3/2
T17/7/0    =   T18/7* -T3/0
T17/7/1    =   T18/7* -T3/1
T17/7/2    =   T18/7* -T3/2
X/R/0/0    =   T17/0/0
X/R/0/1    =   T17/0/1
X/R/0/2    =   T17/0/2
X/R/1/0    =   T17/1/0
X/R/1/1    =   T17/1/1
X/R/1/2    =   T17/1/2
X/R/2/0    =   T17/2/0
X/R/2/1    =   T17/2/1
X/R/2/2    =   T17/2/2
X/R/3/0    =   T17/3/0
X/R/3/1    =   T17/3/1
X/R/3/2    =   T17/3/2
X/R/4/0    =   T17/4/0
X/R/4/1    =   T17/4/1
X/R/4/2    =   T17/4/2
X/R/5/0    =   T17/5/0
X/R/5/1    =   T17/5/1
X/R/5/2    =   T17/5/2
X/R/6/0    =   T17/6/0
X/R/6/1    =   T17/6/1
X/R/6/2    =   T17/6/2
X/R/7/0    =   T17/7/0
X/R/7/1    =   T17/7/1
X/R/7/2    =   T17/7/2
```

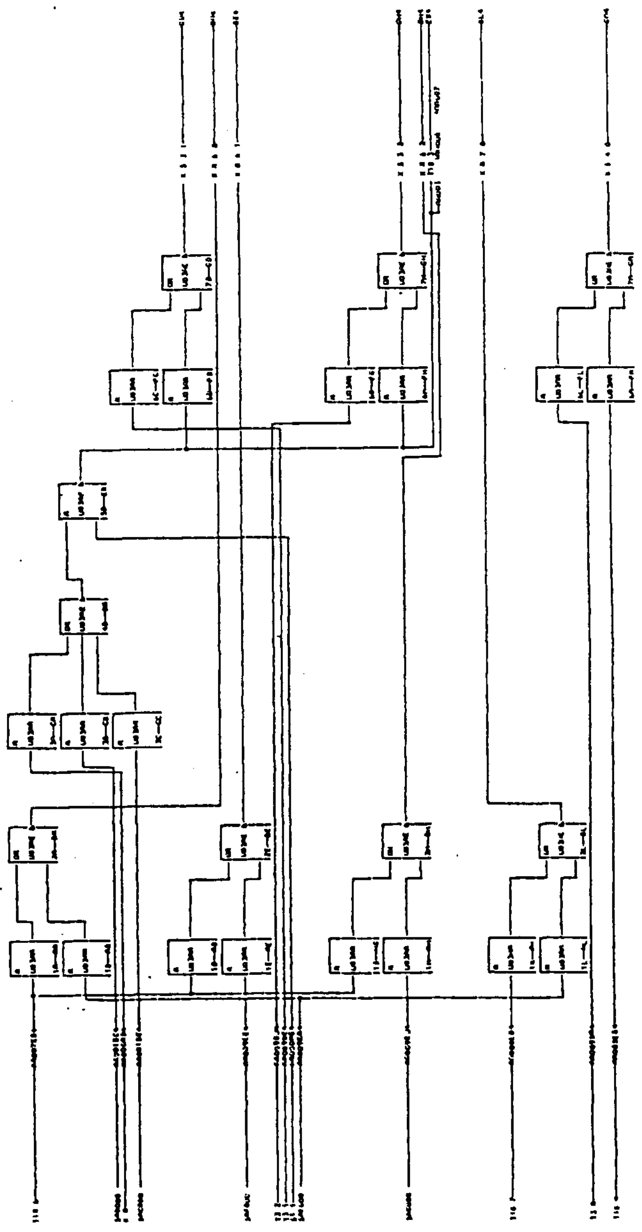
## APPENDIX II

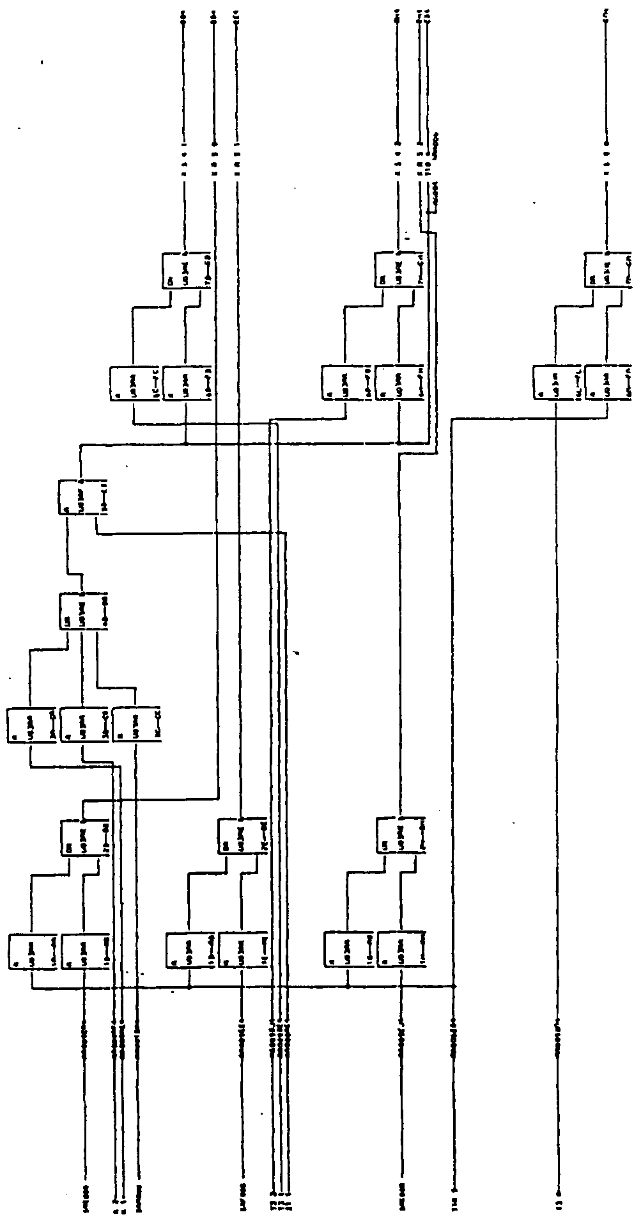
LOGIC SHEETS AUTOMATICALLY DIAGRAMMED BY THE IBM LOGIC AUTOMATION SYSTEM

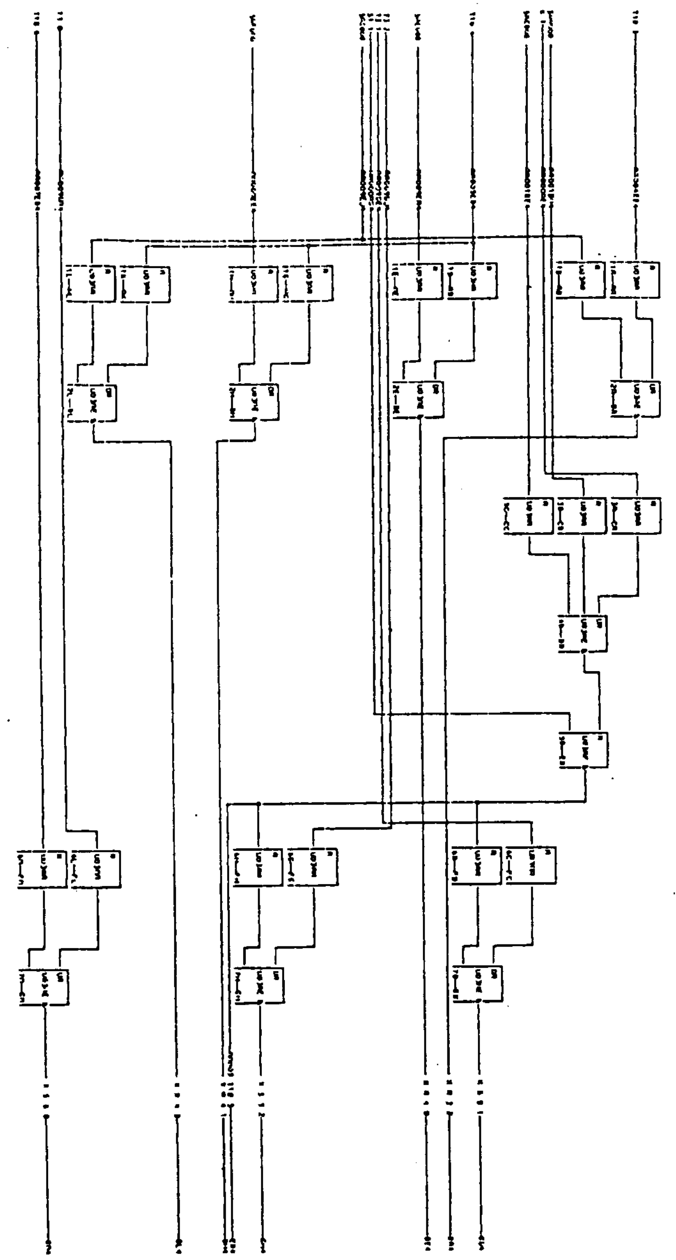


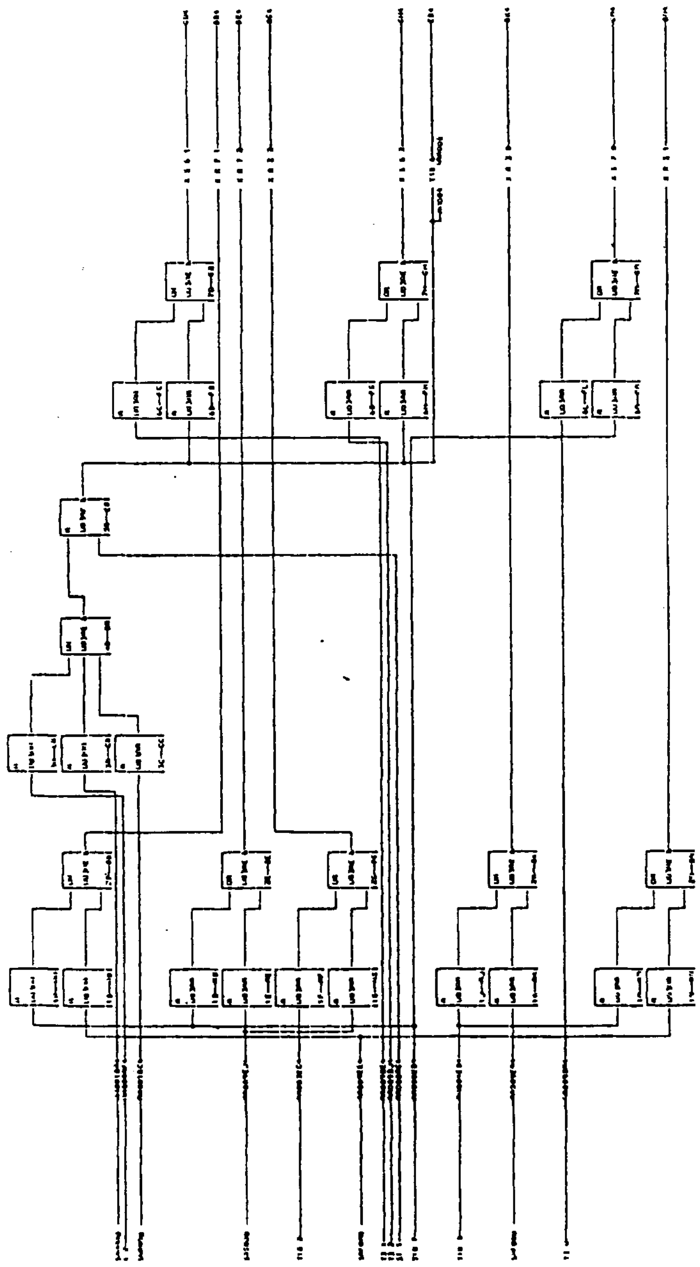


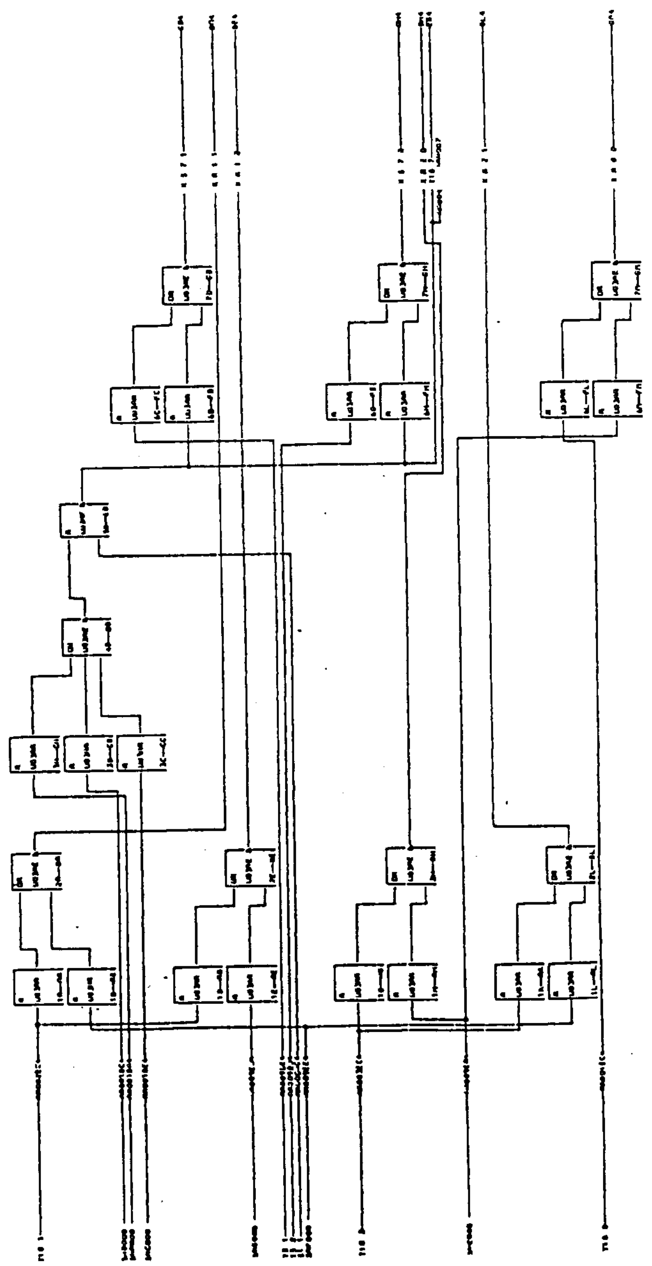


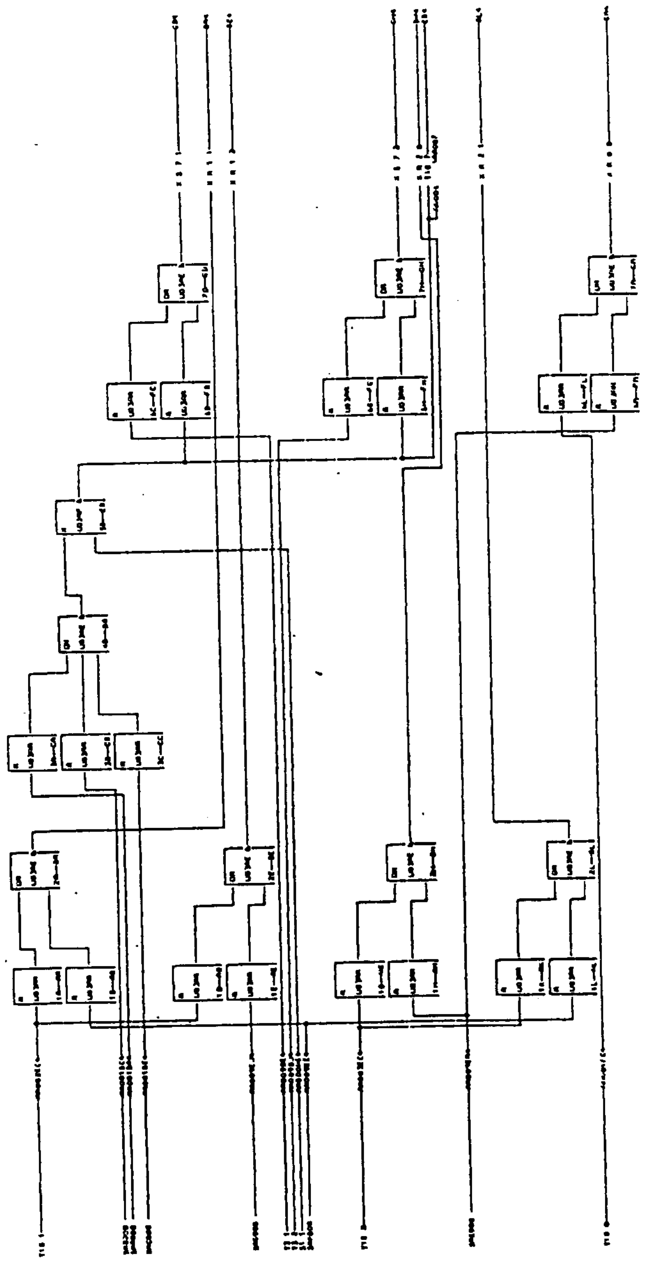












## Acknowled[illegible]ent

The authors wish to th[illegible] [illegible]. D. Coleman, now at IBM Scientific Center, Los Angeles, Calif., for initiating this project.

## References

[1] P. W. Case, H. H. Graff, L. E. Griffith, A. R. LeClercq, W. B. Murley, and T. M. Spence, "Solid logic design automation for IBM system/360," *IBM J. Res. Develop*, vol. 8, pp. 127–140, April 1964.
[2] Y. Chu, "An ALGOL-like computer design language," *Commun. ACM*, vol. 8, pp. 607–615, October 1965.
[3] A. D. Falkoff and K. E. Iverson, "APL\360: User's manual," IBM Watson Research Center, Yorktown Heights, N. Y., 1968.
[4] A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "Formal description of system/360," *IBM Sys. J.*, vol. 3, pp. 198–262, 1964.
[5] T. D. Friedm[illegible] an[illegible] C. Yang, "Quality of designs from an automatic l[illegible] generator," IBM Res. Rept. RC 2068, April 25, 1968. (Copi[illegible] [illegible]ilable on request.)
[6] D. F. Gorma[illegible] and J. P. Anderson, "A logic design translator," *1962 Fall Joint Computer Conf., AFIPS Proc.*, vol. 22. Washington, D. C.: Spartan Books, 1962, pp. 251–261.
[7] K. E. Iverson, *A Programming Language*. New York: Wiley, 1962.
[8] G. Metze and S. Seshu, "Computer compiler, part I—Preliminary report," Coordinated Science Lab., University of Illinois, Urbana, Dept. R-364, August 1965.
[9] J. P. Roth, "Systematic design of automata," *1965 Fall Joint Computer Conf., AFIPS Proc.*, vol. 27, pt. 1. Washington, D.C.: Spartan Books, 1965, pp. 1093–1100.
[10] H. P. Schlaeppi, "A formal language for describing machine logic, timing, and sequencing (LOTIS)," *IEEE Trans. Electronic Computers*, vol. EC-13, pp. 439–448, August 1964.
[11] H. Schorr, "Computer-aided digital system design and analysis using a register transfer language," *IEEE Trans. Electronic Computers*, vol. EC-13, pp. 730–737, December 1964.

11

AS-4

#7

7th DA Conferenc
1970

# QUALITY OF DESIGNS FROM AN AUTOMATIC LOGIC GENERATOR (ALERT)*

Theodore D. Friedman
and
Sih-Chin Yang

IBM Thomas J. Watson Research Center
Yorktown Heights, New York

## Abstract

Automation of the design of computer logic has evoked wide-spread interest and activity. Nevertheless, detailed comparisons of automatically generated logic and manually prepared logic have not been made available.

The ALERT program is a logic generator which accepts as input a summary description of a new computer in a high-level language, and from this it compiles logic designs to carry out the functions specified. This paper examines the quality of logic generated by the ALERT system.

The specifications of several computers have been processed through the ALERT system. This report discusses the most comprehensive design processed, the central processor of the IBM 1800 computer.

The 1800 had been designed by conventional manual techniques prior to this study, and its logic schematics were, therefore, available for comparison with the logic generated by ALERT.

It was found that the automatically produced circuitry required 160% more components than used in the corresponding parts of the actual computer. Reasons for this discrepancy are considered, and methods are described which are expected to reduce the discrepancy between automatically generated designs and manual designs.

The study indicates that automatic generation of computer logic and circuitry from high-level system descriptions offers a practical and viable alternative to traditional methods of logic design. Within the limits of the study, the automatically prepared design was found to be correct, functional, and manufacturable.

## Introduction

There has been wide interest in the possibility of automating the design of computer logic [1], [4], [5], [9], [11], [14], [16]. Feasibility of an automated logic generator was demonstrated by Proctor in 1964 [12], however detailed comparisons of automatically generated logic and manually prepared logic have not been made available. The purpose of this paper is to present such a study.

The ALERT program, [7], [8], is a logic generator programmed to run on the IBM 7094. Logic generators such as ALERT should be distinguished from other design automation programs. Much of the development of new computers has been partially or fully automated, including wire routing, module placement, partitioning, circuit selection, checking, simulation, test generation, documentation and manufacturing [1], [3]. These automated systems, however, still rely on the human designer and engineer to prepare the logic design by hand before they can be used. The purpose of a logic generator is to automate preparation of the logic design itself.

As input to the ALERT program, the designer specifies the characteristics of the computer he desires, and from this the program generates logic equations which realize the specified features. These equations in turn can then be processed by standard Logic Automation [13] and Design Automation [3] programs to produce the circuitry and documentation of the desired computer. ALERT is analogous to a conventional compiler program because it translates its input from a high-level algorithmic form into a detailed machine-oriented form. But rather than generating a program as its output, ALERT generates logic designs for building hardware.

The input is expressed in a form of Iverson's APL notation [10]. To prepare the input, the user specifies the system characteristics of the desired machine, including the registers, memory size, word length or word mark conventions, instruction format and repertoire, indexing and other features. The behavior of the proposed machine is expressed by programs of micro-events.

It was necessary to develop special conventions to depict the machine organization and data flow information. APL had previously been used to describe the programming and functional characteristics of computer systems [6], but the descriptions did not specify the nature of the mechanisms used to provide those characteristics. To employ APL as a design specification language, the most important new convention imposed was for variables in the programs to be considered logic devices such as flip-flops, gate output lines, or registers, while the program statements themselves are interpreted as connections and gating among these devices. Accordingly, variables must be declared to be of fixed sizes and types. New statement types were introduced, such as assignment statements which affect only the set or reset lines of flip-flops (written: ← 'SET', ← 'RESET'). Because the input is punched on cards to be accepted by the 7094, APL symbols are transliterated. An-

* This report is a revision of IBM Research Report RC 2068.

other innovation in this version of APL is the period or statement terminator -- all entries to the right of a period in a card are considered to be comments.

The specifications of several computers were prepared in this way and were processed through the ALERT system. We will discuss the most comprehensive design processed, the central processor of the IBM 1800 computer.

The 1800 was selected because of its moderate size and its currentness. It is a general purpose machine intended for process control applications, and built with the IBM Solid Logic Technology circuit family. The 1800 had been designed and manufactured before this study was undertaken, and its logic schematics were available for comparison with the output logic generated by ALERT.

## The 1800 Input Description

A listing of the input deck processed by ALERT describing the 1800 computer is shown in Appendix I.* This description was derived from the 1800 Systems Manual [15] and various engineering documents. An early description of the machine in Iverson's notation by K. Burgbacher was also helpful [2]. The description in Appendix I includes the major part of the 1800 Processor-Controller unit, but console devices, channels, input and output features, and the initial program loading facility are omitted. Twenty six of the 31 instructions in the machine's repertoire are included.

The description begins with declarations of registers, important flip-flops, and variables. The 1800's accumulator, for example, indicated in the first declaration statement by the name "A," is declared to consist of 16 flip-flops. The first occurrence of a name in the description fixes its size and dimensions. If no dimensions are given, as in the case of ADDFF in the second declaration statement, it is filed as a single bit.

Following the declarations, programs of micro-events are listed. The first two programs are special pre-specified functions (Macro's), as indicated by a card with an "F" in column one. Macro functions are not entered into the design until later during processing when they may be "called" by other programs. Then they are copied and "plugged in" at each point of the design where a call occurred. The first Macro defines the design of decoders and the second defines a digital comparator.

Following these Macro functions are the ordinary programs, each of which is used to generate a section of the final machine. The first of these, MACO, specifies the memory access mechanism in the 1800. Following this are programs for interruption processing (INT), instruction counter advancement (ICNTR), interrupt queuing (LOAD), the adder (ADD), the shift control counter (SCCTR), register transfers (SHIFTL, SHIFTR, SWAP, BTOD, DTOA, ATOU, UTOA, XRTOSC, SCTOXR, XRTOA, and ATOXR). The next program (E) specifies the algorithms for execution of the instruction set, and the last program (CPU) describes the sequencing and control of the entire central processor.

Although space prevents review of the entire design, the reader may be assisted in understanding the computer description and the automatic design process by discussion of a part of the design. In particular, the Shift Control Counter program is examined in Appendices II and III.

## Summary of the Findings

The 1800 description in Appendix I was processed by the ALERT system and by the IBM Logic and Design Automation systems to generate schematic logic diagrams of the machine. The diagrams were produced in a wholly automated fashion. One of these diagrams is shown and discussed in Appendix III.

The resulting design is considerably different in detail from the actual 1800 computer. Although the automatically produced design appeared for the most part correct and manufacturable, some systematic means were needed to evaluate it. The most scrupulous evaluation would have involved manufacturing the machine and then comparing its cost and performance with the real 1800. However such an approach would be excessively expensive. Moreover, a variety of flaws and inadequacies were discovered in the automatically produced design which would require correction before it could be built. (The flaws were generally minor and readily correctable, resulting from bugs in the ALERT programs or faults in the computer description. However, the 7094 computer used for running our study became unavailable before we could make those corrections.)

Rather than manufacturing the machine, therefore, we felt that the next best method for evaluation would be to compare the numbers of components used in the real machine and in the automatically produced design. There were difficulties in this approach as well since the circuits implemented by the Logic Automation System differed from those used in the real 1800. Nevertheless, we made a circuit-by-circuit comparison of the two designs, checking the validity of the automatic design as we did so. The results are presented in Table I. Much of this table is concerned with various anomalies in the automatically produced designs, as described below.

Column 1 of the table identifies sections into which the 1800 design was partitioned. Partitioning was required because the design exceeded available memory in the 7094 computer and space limitations in the Logic Automation program. This partitioning, in turn, led to redundancies and incon-

* Some familiarity with the conventions of Iverson's APL Notation is desirable for proper understanding of this computer description. The reader is referred to the summary on pp. 195-203 of [6].

TABLE I

SUMMARY OF LOGIC BLOCK
COUNTS FROM ALERT AND MANUAL DESIGNS

| (1) Section of 1800 Design | (2) Number of Logic Blocks Produced by ALERT | (3) Number of Blocks after Reprocessing for circuitry | (4) Correction for Partitioning Effect | (5) Correction for Clocking Control | (6) Correction for Flip-Flops | (7) Corrected Total from ALERT | (8) Corresponding Sections of Real 1800 | (9) Discrepancy of ALERT Total over Real 1800 |
|---|---|---|---|---|---|---|---|---|
| Memory Access | 147 | 424 | -32 | +32 | +25 | 449 | 231 | 218 |
| Interruption Processor | 675 | 1004 | -27 | +19 | +108 | 1104 | 505 | 599 |
| Instruction Counter Control | 128 | 239 | — | — | +16 | 255 | 92 | 163 |
| Interrupt Queuing | 60 | 149 | -27 | — | — | 122 | 62 | 60 |
| Adder | 496 | 1173 | -31 | +29 | +20 | 1171 | 521 | 650 |
| Shift Control Counter | 46 | 102 | - 6 | — | — | 96 | 36 | 60 |
| Register Transfer | 1043 | 1750 | -184 | — | +80 | 1646 | 587 | 1059 |
| Instruction Execution | 961 | 1598 | -198 | +285 | — | 1685 | 554 | 1131 |
| Processor Control | 934 | 1579 | -207 | +115 | +18 | 1505 | 484 | 1021 |
| TOTAL | 4372 | 8018 | -712 | +480 | +267 | 8033 | 3072 | 4961 |

TABLE II

SUMMARY OF DISCREPANCIES

| | | |
|---|---|---|
| | Corrected output from ALERT | 8033 logic blocks |
| | Size of corresponding sections of real 1800 | 3072 |
| | Discrepancy of ALERT over real design | 4961 or 160% more blocks from ALERT than in real design |
| (minus) | Estimated reduction by modifying circuit implementation rules | 2357 |
| (minus) | Estimated Reduction by revising input description | 568 |
| (minus) | Estimated reduction by system improvements | 1020 |
| | Theoretical Discrepancy | 1016, or 33% more blocks from ALFRT than in real design |

plete coordination of the sections of the design, which are discussed later.

Column 2 lists the number of logic blocks produced by ALERT for each section. A logic block is a circuit which performs a logical function such as an AND gate, an OR gate, or an inverter. The output of ALERT represents idealized logic blocks, since circuit restrictions were not considered in producing these results. A total of 4372 idealized logic blocks were used in the design.

Column 3 lists the number of logic blocks which resulted after the idealized logic was automatically converted into actual circuitry by the IBM Logic Automation and Design Automation programs. In this conversion the designs from ALERT were reprocessed to cause them to conform to circuit constraints such as fan-in, fan-out, and module pin restrictions. This conversion caused extra logic blocks to be added into design to satisfy the engineering constraints: The total logic block count was increased from 4372 to 8018.

Column 4 shows the effects on block counts attributed to partitioning of the Design File.

The partitions segregated parts of the design that were functionally related. This led to the following anomalies:

(a) It was impossible to combine logic gates which have identical inputs if they appear in different sections.

(b) Redundant inverters were sometimes used. For example, suppose a signal, X, and its complement, $\overline{X}$, are available in the Adder Section, but only the X signal is represented in the CPU section. If $\overline{X}$ is also needed in the CPU a new inverter will automatically be generated instead of obtaining the complemented signal from the Adder. Redundancy also occurs in the case of extra gates supplied for driving loads.

(c) OR gates located in distinct sections and which have the same output destinations were not combined.

The figures in Column 4 were derived manually by counting instances of these anomalies.

The partitioning effects could be reduced or eliminated if larger storage capacity were available for ALERT and if storage restrictions in the Logic Automation program could be relieved.

Column 5 of Table I reports counts of logic blocks required for phase counters and their decoding networks. The decoded outputs are used to form timing control signals. In the ALERT system, the sizes of phase counters are determined automatically, however the logic elements needed to construct them were omitted from the final output. To correct this fault, these missing elements were manually added to the total logic count.

It may be noted that certain sections have no entry in Column 5. These sections correspond to programs with manually specified control signals which are generated elsewhere -- these sections consequently do not require phase counters. A total of 480 logic blocks are estimated for the phase counters.

Column 6 lists the number of flip-flops required. The Logic Automation program cannot represent flip-flops unless its input data is specified in Injective Word Format [13]. Since the results from ALERT were not produced in that format, flip-flops were not represented in the output logic drawings and it was necessary to supply them manually.

A given flip-flop may be used by several sections, but it is counted only at its initial use.

Column 7 shows the block counts from ALERT corrected for the partitioning effect, clocking control and flip-flops. This represents the sum of columns 3, 4, 5, and 6, providing a total of 8033 blocks.

Column 8 indicates the component counts used in the corresponding sections of the real IBM 1800. These figures were obtained by examination of the 1800 engineering manuals. A total of 3072 logic blocks were counted.

Column 9 indicates the discrepancy in logic block count from ALERT in column 7 over the real design. A total discrepancy of 4961 logic blocks was found, that is, 160% more logic blocks were generated by ALERT than were used in corresponding sections of the manual design.

## Projected Improvements

The discrepancy between ALERT and the actual design can be reduced by the following methods, summarized in Table II.

(a) Extensions of the Choice of Circuit Types in the Logic Automation Programs

The ALERT logic equations were implemented using only three types of circuits, namely, AI (AND Inverter), AOI (AND-OR Inverter) and N (Single Input Inverter with driving power). These contrast with dozens of circuit types used in the actual design. Because of this, more logic blocks were generated than would be if there were a greater choice of circuits. This situation is compounded by the partitioning effect as well as by input and system deficiencies (see below). A total of 2357 logic blocks are attributed to this cause.

(b) Revision of the 1800 Description

Examination of the output logic indicated that if certain input statements were respecified or regrouped, the number of logic blocks required could be reduced without changing the intended functions. For example, saving any sometimes be achieved if a register is pre-cleared before data is gated into it. Saving would also result if common signals would be combined as illustrated in Figure 1. In that case, the four instances of the input signal "3" in Figure 1(a) could be replaced by one input in Figure 1(b). This saving, however, depends on the types of circuits available for implementing the logic. Another saving could be realized by rearranging input statements to reduce the number of timing phases in a program. There are several other techniques as well that would result in savings. It is estimated that 568 logic blocks can be eliminated from the output if all these input revisions would be incorporated.

(c) System Improvements

A number of improvements in the ALERT system have been projected although they have not been implemented. One improvement is sharing phase counters. Several programs could share a common counter to reduce the number of counters required. Also, within a single program, statements may be divided into groups and the phase counter value reset to one at the beginning of each group. This approach would reduce the size of the counter and the number of its decoded outputs. However, more control signals would then be needed to distinguish each group.

Another possible improvement would involve greater coordination of array inputs. For example, assume A, B, C, D are all 4 bit vectors. The statement

A = B 'AND' C 'AND' D

will presently produce the gating shown in Figure 2(a). The improved version should produce the structure shown in Figure 2(b).

Still another improvement would be to cause ALERT to generate output in the Injective Word Format used in the Logic Automation System. This would avoid difficulties such as the limited circuit choice. Various other improvements are also projected, but are too detailed to list here. These improvements, it is estimated, would save a total of 1020 logic blocks in the 1800 design.

After all corrections, the automatically generated logic would then exceed the manual design by 1016 blocks. (See Table II.)

## Conclusions

The current study suggests that automatic generation of computer logic and circuitry from high-level system descriptions offers a practical and viable alternative to traditional methods of logic design. Apart from a number of remediable errors, the automatically prepared design was found to be correct, functional, and manufacturable. Although the automatically produced circuitry required more than twice as many components as used in the corresponding manual designs, this discrepancy is expected to be reduced by projected system improvements.

It is anticipated that compilers of computer logic will provide some of the benefits that conventional compilers have provided to programmers. Automated techniques should not only aid development directly, but should facilitate checking, rework and documentation.

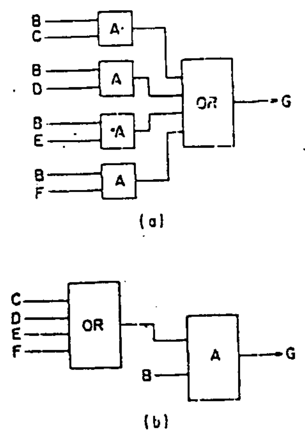


Fig. 1. Combining common signals.

GIVEN A, B, C, D, ALL 4 BIT VECTORS; THEN
A = B 'AND' C 'AND' D
WILL PRODUCE

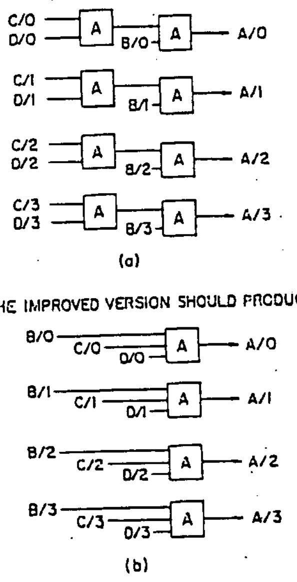


Fig. 2. Coordination of array signals.

## APPENDIX I

```
*    THIS IS A DESCRIPTION OF THE [illegible] COMPUTER
*
*              NOTE.. ANYTHING TO THE RIGHT OF A PERIOD IS A COMMENT
*
*              THE FOLLOWING STATEMENTS ARE DECLARATIONS,
*                          AS INDICATED BY A 'D' IN COLUMN 1.
*
D     [illegible].             ACCUMULATOR (DECLARED TO BE 16 FLIP-FLOPS).
D     [illegible].             ADD/SUB BIT
D     [illegible].             STORAGE BUFFER REGISTER (CALLED B)
D     [illegible].             B-REGISTER IS BEING LOADED FROM CORE
D     [illegible].             BRANCH [illegible] BIT
D     [illegible].             CARRY BIT
D     [illegible].             COMBINATION SIGNAL OF [illegible]
D     [illegible].             ARITHMETIC FACTOR REGISTER
D     [illegible].             DOUBLE PRECISION CARRY
D     [illegible].             SIGN COMPARISON BIT FOR ARITHMETIC OF
*                                 [illegible] IF OPERANDS HAVE UNLIKE SIGNS)
D     DISABL.                   DISABLE INTERRUPT SWITCH
D     [illegible].             FORMAT REGISTER (ALSO CALLED FLAG)
*                                 ([illegible] WORD [illegible], [illegible] WORD [illegible])
D     [illegible].             CURRENT INTERRUPT LEVEL (CALLED [illegible])
D     [illegible].             HIGHEST PRIORITY INTERRUPT SIGNAL
D     [illegible].             INDIRECT ADDRESSING BIT
D     [illegible].             INHIBIT DRIVERS TO CORE WINDINGS
D     [illegible].             INTERRUPT LEVEL REGISTER
D     [illegible].             INTERRUPT LEVEL STATUS WORD
D     [illegible].             STOP CONTROL FLAG
D     [illegible].             INTERRUPT MASK REGISTER
D     [illegible].             DECODED OUTPUT OF [illegible]
D     [illegible].             MEMORY CONTROL FLAGS
*                                  [illegible] [illegible] CORE [illegible] CYCLE [illegible]
*                                  [illegible] [illegible] ANOTHER [illegible] ACTIVE
*                                  [illegible] [illegible] IN SINGLE STORAGE CYCLE
*                                  [illegible] [illegible] SINGLE-[illegible] CYCLE [illegible]
*                                      [illegible] START [illegible]
D     [illegible].             UTILITY VARIABLE FOR [illegible] FUNCTION
D     [illegible].             LATCH [illegible] IS LAST BIT OF ADDRESS
D     [illegible].             [illegible] SWITCH
D     NAV1.                     NAVIGATION VECTOR 1
D     NAV2.                     NAVIGATION VECTOR 2
D     [illegible].             NAVIGATION VECTOR 3
D     [illegible].             NAVIGATION VECTOR 4
D     [illegible].             [illegible] REGISTER
D     [illegible].             [illegible] REGISTER
D     [illegible].             OPERATING STATE
*                                  [illegible]
D     [illegible].             PARITY BIT
D     [illegible].             PROGRAM SELECTION
D     [illegible].             ACCUMULATOR EXTENSION
D     [illegible].             SENSE AMPLIFIERS FROM CORE WINDINGS
*                                  [illegible] STORAGE [illegible],
*                                  [illegible] PARITY, [illegible] PARITY.
D     [illegible].             STORAGE ADDRESS REGISTER

*                               [illegible] CALLED [illegible] REGISTER)
D     [illegible].             SHIFT COUNTER
D     [illegible].             TEMPORARY SUM
D     [illegible].             INTERRUPT REQUEST REGISTER
D     [illegible].             TAG TO SELECT INDEX REGISTER
D     [illegible].             TEMPORARY ACCUMULATOR REGISTER
D     [illegible].             STORAGE PROTECTION BIT
D     [illegible].             [illegible] CONTROL
*                               [illegible]
D     [illegible].             DECODER FOR X-DRIVERS OF CORE WINDINGS
D     [illegible].             [illegible] ADDRESS REGISTER
*                                   (ALSO CALLED I REGISTER)
*                               [illegible] REGISTERS
D     [illegible].             DECODER FOR Y-DRIVERS OF CORE WINDINGS
*
*
*           NOTE.. MICRO-PROGRAMS ARE INDICATED BY AN 'M' IN COLUMN 1.
*                 FUNCTIONS ARE MICRO-PROGRAMS WHICH MAY BE PLUGGED
*                       INTO OTHER PROGRAMS.  FUNCTIONS ARE INDICATED
*                       BY AN 'F' CARD, WHICH SHOWS THE CALLING FORMAT
*                       USING DUMMY ARGUMENTS
*
M DECODE.          DECODE FUNCTION                                          DECOD 6
*                                                                           DECOD 01
F[illegible]   [illegible] = 'DECODE' INPUT.                                DECOD 03
*                                                                           DECOD 05
      SELECT SEQUENCE [illegible] = 'AND'/ SEQUENCE '[illegible]' 'VALUE' INPUT.   DECOD 1
*
*
*
M [illegible].            INTERRUPT LEVEL COMPARISON FUNCTION               [illegible]
*                                                                           [illegible]
F[illegible]      [illegible] = [illegible] '[illegible]' [illegible].      [illegible]
*                                                                           [illegible]
          [illegible] = 'AND'/[illegible],                                  [illegible]
                              'AND'[illegible].                             [illegible]
*
*
*
M [illegible].    .    MEMORY ACCESS FOR PROCESS-CONTROLLER (MAC NUMBER [illegible])   MACD [illegible]
*                                                                                   MACD [illegible]
          [illegible] = 1.                                                          MACD [illegible]
  [illegible]..  IF ( 'AND'/ ( [illegible] 'AND' [illegible] ) ) [illegible]        MACD [illegible]
                 [illegible] GO TO [illegible].                                     MACD [illegible]
          [illegible] = 1.                                  SET INTERLOCKS         MACD 3
          [illegible] = ( [illegible] ).                    [illegible] X, Y-DRIVERS   MACD [illegible]
          [illegible] = ( [illegible] ), [illegible].                               MACD [illegible]
          [illegible] = [illegible].                        READ-OUT CYCLE          MACD 5
          [illegible] = 0.                                  [illegible] AVAILABLE   MACD 6
          [illegible] = '[illegible]'.                                              MACD 7
          [illegible] = '[illegible]'.                      CHECK [illegible] STORE OF  MACD [illegible]
          [illegible] = '[illegible]'.                      IS THIS A STORE [illegible]  MACD 11
          [illegible] = '[illegible]'.                      IF [illegible] RETURN [illegible]  MACD 12
          [illegible] = '[illegible]'.                                              MACD 13
```

77

DEF012011

```
[illegible]
* INP.        INTERRUPT PROCESSING
[illegible]
              FORCE OPERATING STATE TO [illegible]
[illegible]
* ICNTR.      INSTRUCTION COUNTER ADVANCEMENT FACILITY
[illegible]
[illegible]
* ADD.        ADD AND SUBTRACT FACILITY
[illegible]
```

DEF012012

```
                 ('AND'/SUF(0),'NOT'SUF(1,2))'OR'('AND'/MAXB)).      ADDN  22
      APP = 0.                         RESET AND PROGRAM FLIP-FLOP   ADDN  23
      STOP.                                                          ADDN  24
*
** SCCIN.     SHIFT COUNTER INCREMENT FACILITY                       SCCIN 0
*                                                                    SCCIN 01
      SC =   'SET' STAPSC 'AND' ('AND'/'NOT'SC),                     SCCIN 02
                                ('AND'/'NOT'SUF(1)(5)/SC),           SCCIN 04
                                ('AND'/'NOT'SUF(1)(4)/SC),           SCCIN 06
                                ('AND'/'NOT'SUF(1)(3)/SC),           SCCIN 08
                                ('AND'/'NOT'SUF(1)(2)/SC),           SCCIN 10
                                 'NOT'SC(5).                         SCCIN 12
*                                                                    SCCIN 13
      SC = 'RESET' STAPSC 'AND' ('AND'/SC(0),'NOT'SUF(1)(5)/SC),     SCCIN 14
                                ('AND'/SC(1),'NOT'SUF(1)(4)/SC),     SCCIN 16
                                ('AND'/SC(2),'NOT'SUF(1)(3)/SC),     SCCIN 18
                                ('AND'/SC(3),'NOT'SUF(1)(2)/SC),     SCCIN 20
                                ('AND'/SC(4),'NOT'SC(5)),            SCCIN 22
                                 SC(5).                              SCCIN 24
*
*
** SHIFTL.    SHIFT LEFT FACILITY FOR A, B REGISTERS                 SHFTL 0
*                                                                    SHFTL 01
      A =   'SET' ((SHLA'OR'SHLAB) 'AND' SUF(1)(1)/A),               SHFTL 02
                  ('AND'/SHLAB, B(0)).                               SHFTL 03
      B = 'RESET' ((SHLA'OR'SHLAB) 'AND' 'NOT'SUF(1)(1)/A),          SHFTL 05
                  ('OR'/('AND'/SHLAB, 'NOT'B(0))), SHLA.             SHFTL 06
*                                                                    SHFTL 08
      B =   'SET' SHLAB 'AND' SUF(1)(1)/B, 'NOT'1A.                  SHFTL 10
      B = 'RESET' SHLAB 'AND' ('NOT'SUF(1)(1)/B), 1A.                SHFTL 11
*
*
** SHIFTR.    SHIFT RIGHT FACILITY FOR A, B REGISTERS                SHFTR 0
*                                                                    SHFTR 01
      A =   'SET' (SHRA'OR'SHRAB) 'AND' ('AND'/1A,ARI,A(1)),         SHFTR 03
                                        PRE(1)(1)/A).                SHFTR 04
      A = 'RESET' (SHRA'OR'SHRAB) 'AND' ('AND'/1A,ARI,'NOT'A(1)),    SHFTR 05
                                        'NOT'PRE(1)(1)/A).           SHFTR 06
*                                                                    SHFTR 08
      B =   'SET' SHRAB 'AND' A(15),PRE(1)(1)/B.                     SHFTR 10
      B = 'RESET' SHRAB 'AND' ('NOT'A(15), 'NOT'PRE(1)(1)/B.         SHFTR 12
*
*
** SWAP.      EXCHANGE A WITH B FACILITY                             SWAP  0
*                                                                    SWAP  01
      A =   'SET' SWAB 'AND' B.                                      SWAP  03
      A = 'RESET' SWAB 'AND' 'NOT'B.                                 SWAP  04
      B =   'SET' SWAB 'AND' A.                                      SWAP  06
      B = 'RESET' SWAB 'AND' 'NOT'A.                                 SWAP  07
*
*
** BTOW.      LOAD B WITH W FACILITY                                 BTOW  0
*                                                                    BTOW  01
      B =   'SET' LDABW 'AND' W.                                     BTOW  03
      B = 'RESET' LDABW 'AND' 'NOT'W.                                BTOW  05
*
*
** WTOA.      LOAD A WITH W FACILITY                                 WTOA  0
                                                                     WTOA  0
      A =   'SET' (LWAAW2'AND'W) 'OR' (LWAAW3'AND'(W'AND''NOT'A)).   WTOA  03
      A = 'RESET' (LWAAW3'AND''NOT'W) 'OR' (LWAAW3'AND'(W'AND'A).    WTOA  05
*
*           LWAAW1,LWAAW2,LWAAW3 ARE CONTROL SIGNALS FOR             WTOA  06
*           'AND' 'OR' AND 'EOR' RESPECTIVELY                        WTOA  07
*
*
** ATOW.      STORE ACCUMULATOR IN W                                 ATOW  0
*                                                                    ATOW  01
      W =   'SET' LDAW 'AND' A.                                      ATOW  03
      W = 'RESET' LDAW 'AND' 'NOT'A.                                 ATOW  05
*
*
** WTOA.      ADDING ACCUMULATOR WITH W                              WTOA  0
*                                                                    WTOA  01
      A =   'SET' ADING 'AND' W.                                     WTOA  03
      A = 'RESET' ADING 'AND' 'NOT'W.                                WTOA  05
*
*
** SKTOSC.    LOAD SC WITH SK FACILITY                               SKTOSC 0
*                                                                    SKTOSC 01
      SK(P(1)(N)=0) = LDSKSC 'AND' SUF(1)(3)/INDEX.                  SKTOSC 03
      SC =   'SET' (SK(P(0)) 'AND' SUF(1)(N)/SK(1)) 'OR'             SKTOSC 05
                   (SK(P(1)) 'AND' SUF(1)(N)/SK(2)) 'OR'             SKTOSC 06
                    SK(P(2)) 'AND' SUF(1)(N)/SK(3)).                 SKTOSC 07
      SC = 'RESET' (SK(P(0)) 'AND' 'NOT'SUF(1)(N)/SK(1)) 'OR'        SKTOSC 10
                   (SK(P(1)) 'AND' 'NOT'SUF(1)(N)/SK(2)) 'OR'        SKTOSC 11
                    SK(P(2)) 'AND' 'NOT'SUF(1)(N)/SK(3)).            SKTOSC 12
*
*
** SCTOSK.    LOAD LAST 6 BITS OF SK WITH SC                         SCTOSK 0
*                                                                    SCTOSK 01
      SC(P(1)(N)=0) = LDSCSK 'AND' SUF(1)(3)/INDEX.                  SCTOSK 03
      SUF(1)(N)/SK(1) =   'SET' SC(P(0)) 'AND' SC.                   SCTOSK 04
      SUF(1)(N)/SK(2) =   'SET' SC(P(1)) 'AND' SC.                   SCTOSK 06
      SUF(1)(N)/SK(3) =   'SET' SC(P(2)) 'AND' SC.                   SCTOSK 07
      SUF(1)(N)/SK(1) = 'RESET' SC(P(0)) 'AND' 'NOT'SC.              SCTOSK 08
      SUF(1)(N)/SK(2) = 'RESET' SC(P(1)) 'AND' 'NOT'SC.              SCTOSK 09
      SUF(1)(N)/SK(3) = 'RESET' SC(P(2)) 'AND' 'NOT'SC.              SCTOSK 10
*
*
** SKTOA.     LOAD A WITH SK FACILITY                                SKTOA 0
*                                                                    SKTOA 01
      SK(P(1)(N)=0) = FILLA 'AND' SUF(1)(3)/INDEX.                   SKTOA 03
      A =   'SET' (SK(P(0))'AND'SK(1)) 'OR'                          SKTOA 05
                  (SK(P(1))'AND'SK(2)) 'OR'                          SKTOA 06
                   SK(P(2))'AND'SK(3)).                              SKTOA 07
      A = 'RESET' (SK(P(0))'AND''NOT'SK(1)) 'OR'                     SKTOA 10
                  (SK(P(1))'AND''NOT'SK(2)) 'OR'                     SKTOA 11
```

```
                [illegible]                                          [illegible] 13
*
*
[illegible].       LOAD AA WITH A FACILITY                          ALU44 0
*                                                                   ALU44 01
          [illegible] = [illegible] 'AND' [illegible].              ALU44 02
          [illegible] = '[illegible]' [illegible] 'AND' A.          ALU44 03
          [illegible] = '[illegible]' [illegible] 'AND' A.          ALU44 06
          [illegible] = '[illegible]' [illegible] 'AND' A.          ALU44 07
          [illegible] = '[illegible]' [illegible] 'AND' 'NOT'A.     ALU44 08
          [illegible] = '[illegible]' [illegible] 'AND' 'NOT'A.     ALU44 09
          [illegible] = '[illegible]' [illegible] 'AND' 'NOT'A.     ALU44 10
*
*
          [illegible].
*
*
*
* 4.          INSTRUCTION EXECUTION                                 4     0
*                                                                   4     01
          GO TO [illegible].                                        4     02
*
*
*DOUBLE LOAD INSTRUCTION       'LDD'                                4     09
  [illegible].. [illegible] = 1.       PREPARE FOR DOUBLE WORD INST 4     10
          [illegible] = 1.             FETCH WORD FROM MEMORY       4     12
  [illegible].. IF [illegible] GO TO [illegible].  WAIT TILL B IS AVAILABLE  4  13
          [illegible] = 1.             LOAD B WITH A                4     14
  [illegible].. [illegible] = 1.       LOAD A WITH M                4     15
  [illegible].. IF [illegible] GO TO [illegible].                   4     16
          [illegible] = 0.                                          4     16
          [illegible] = 1.             FETCH SECOND WORD            4     17
  [illegible].. IF [illegible] GO TO [illegible].                   4     18
          [illegible] = 1.                                          4     19
          [illegible] = 1.             EXCHANGE A WITH B            4     20
  [illegible].. [illegible] = 1.                                    4     23
          GO TO FINISH.                                             4     24
*
*
*LOAD ACCUMULATOR INSTRUCTION   'LD'                                4     29
  [illegible]..                                                     4     30
          [illegible] = 1.                                          4     31
  [illegible].. IF [illegible] GO TO [illegible].                   4     32
          [illegible] = 1.                                          4     33
          [illegible] = 1.                                          4     34
          GO TO FINISH.                                             4     36
*
*
*DOUBLE STORE INSTRUCTION      'STD'                                4     39
  [illegible].. [illegible] = 1.                                    4     40
          [illegible] = 1.                                          4     41
          [illegible] = 1.                                          4     43
          [illegible] = 1.                                          4     44
  [illegible].. IF [illegible] GO TO [illegible].                   4     45
          [illegible] = 0.                                          4     49
          [illegible] = 1.                                          4     50
          [illegible] = 1.                                          4     54
          GO TO FINISH.                                             4     55
*
*
*STORE ACCUMULATOR INSTRUCTION  'STO'                               4     59
  [illegible]..                                                     4     60
          [illegible] = 1.                                          4     61
          [illegible] = 1.                                          4     62
          GO TO FINISH.                                             4     63
*
*
*ADD AND SUBTRACT INSTRUCTIONS  'A' AND 'S'                         4     69
  [illegible]..                                                     4     70
          [illegible] = 1.                                          4     72
  [illegible].. IF [illegible] GO TO [illegible].                   4     73
          [illegible] = 1.                                          4     74
          [illegible] = [illegible].                                4     75
          [illegible] = 1.                                          4     76
          GO TO FINISH.                                             4     77
*
*
*DOUBLE ADD AND SUBTRACT        'AD' AND 'SD'                       4     79
  [illegible]..                                                     4     81
          [illegible] = 1.                                          4     82
          [illegible] = 1.                                          4     83
  [illegible].. IF [illegible] GO TO [illegible].                   4     84
          [illegible] = 1.                                          4     85
          [illegible] = 1.                                          4     87
          [illegible] = [illegible].                                4     90
          [illegible] = 1.                                          4     92
  [illegible].. IF [illegible] GO TO [illegible].                   4     93
          [illegible] = [illegible].                                4     94
  [illegible].. IF [illegible] GO TO [illegible].                   4     95
          [illegible] = 0.                                          4     96
          [illegible] = 1.                                          4     97
  [illegible].. IF [illegible] GO TO [illegible].                   4     97
          [illegible] = 1.                                          4     98
          [illegible] = 1.                                          4     99
          [illegible] = 1.                                          4     103
          GO TO FINISH.                                             4     104
*
*
*SHIFT LEFT INSTRUCTIONS   'SLA', 'SLT', 'SLC' AND 'SLCA'           4     110
  [illegible]..                                                     4     112
          IF [illegible] GO TO [illegible].   SKIP AB TO SC IF [illegible]  4  114
          [illegible] = 1.                    LOAD SC WITH [illegible]      4  115
  [illegible].. IF [illegible] GO TO [illegible].                   4     116
  [illegible].. IF [illegible] GO TO FINISH.                        4     117
          GO TO [illegible].                                        4     118
  [illegible].. [illegible] = '[illegible]' [illegible] 'AND' [illegible].   4  120
          [illegible] = '[illegible]' [illegible] 'AND' [illegible].         4  121
          [illegible]. [illegible] = 1.       [illegible] DECREMENTS SC      4  122
```

30

DEF012014

```
            SHLA = SHLIP 'AND' 'NOT'IA.      SIGNAL FOR SHIFTING A LEFT
            SHLAO = SHLIP 'AND' IA.          SIGNAL FOR SHIFTING A,O LEFT
   [illegible].. IF ([illegible]/SC) 'AND' [illegible] GO TO [illegible].
            IF [illegible] GO TO FINISH.
   [illegible].. LOADA = 1.                   LOAD ON WITH SC
            CAADF = [illegible]/SC.
            SC = 0.
            GO TO FINISH.
*
*
*SHIFT RIGHT INSTRUCTIONS      'SHR', 'SHL' AND 'ASR'
  PROC(13)..
            IF [illegible] GO TO [illegible].
            LOADSC = 1.
   [illegible].. IF ([illegible]/SC) GO TO FINISH.
   [illegible].. MAIA = [illegible]/SC. FOR SC=1A ON AO
            SHRIP = 1.
            SHRA = MAIA 'AND' SHRIP.       INTERCHANGE A AND O IF MAIA
            SC = 'RESET' MAIA.
            STEPSC = SHRIP 'AND' 'NOT'MAIA.
            SHRA = SHRIP 'AND' 'NOT'IA.     SHIFT A RIGHT
            SHRAO = [illegible] 'NOT'MAIA.    SHIFT A,O RIGHT
   [illegible].. IF ([illegible]/SC) GO TO [illegible].
            SC = 0.
            GO TO FINISH.
*
*LOAD STATUS INSTRUCTION       'LDS'
  PROC(14)..
            CARRY = [illegible].
            LINK = [illegible].
            STOP.
*
*
*STORE STATUS INSTRUCTION      'STS'
  PROC(15)..
            A = 0.
            A(1) = 'SET' CARRY.
            A(15) = 'SET' LINK.
            CARRY,LINK = 0.
            B = A.
            ML = 1.
            MACW = 1.
   [illegible].. IF [illegible] GO TO [illegible].
            GO TO FINISH.
*
*
*LOAD INDEX INSTRUCTION '      'LDX'
  PROC(17)..
            MACW = 1.
   [illegible].. IF [illegible] GO TO [illegible].
            LOADB = 1.                      LOAD B WITH O
   [illegible].. LOADX, LOADA = 1.            LOAD A WITH O
            IF [illegible] GO TO [illegible].
            FILLX = 1.                      LOAD OA WITH A
            GO TO FINISH.
   [illegible].. XREG = A.
            GO TO FINISH.
*
*
*STORE INDEX INSTRUCTION       'STX'
  PROC(11)..
            IF [illegible] GO TO [illegible].
            FILLA = 1.                      LOAD A WITH OA
            B = A.
   [illegible].. B = 'SET' [illegible] 'AND' [illegible].
            B = 'RESET' [illegible] 'AND' 'NOT'[illegible].
            ML = 1.
            MACW = 1.
   [illegible].. IF [illegible] GO TO [illegible].
            GO TO FINISH.
*
*
*MULTIPLY INSTRUCTION          'M'
  PROC(20)..
            MACO = 1.
            LOAD = 1.
   [illegible].. IF [illegible] GO TO [illegible].
            RO = M(15).
            SC = 16.
            A = 0.
            LOADO = 1.
   [illegible].. IF ([illegible]) GO TO [illegible].
            SHRAO, STEPSC = 1.              SHIFT A,O RIGHT AND STEP SC
            ADD = 0.
   [illegible].. IF ([illegible]/SC) GO TO [illegible].
            GO TO FINISH.
   [illegible].. LOADO = 1.
            IF [illegible] 'OR' [illegible]
                      GO TO [illegible].      IS ADD NECESSARY.
            ADDOF = [illegible] 'OR'
                  [illegible].
            ADD = 1.
   [illegible].. IF (ADD) GO TO [illegible].
            RO = 'NOT' ADD.                 STORE = OR - FOR NEXT TEST
   [illegible].. SHRAO, STEPSC = 1.
   [illegible].. IF ([illegible]/SC) GO TO [illegible].
            GO TO FINISH.
*
*
*LOGICAL AND, LOGICAL OR AND EXCLUSIVE OR INSTRUCTIONS  'AND''OR''XOR'
  PROC(24)..PROC(25)..PROC(26)..
            MACO = 1.
   [illegible].. IF [illegible] GO TO [illegible].
            [illegible] = 1.
            [illegible] = [illegible].           AND A,O TO A
            [illegible] = [illegible].           OR A,O TO A
            [illegible] = [illegible].           XOR A,O TO A
            GO TO FINISH.
```

```
[illegible]
```

## Appendix II

The 1800 performs left and right shifts of its A register (or of the A and Q registers together for extended shifts). The shift instruction algorithms are represented in programs 2 and 3 of the instruction execution ("E") program. According to these algorithms, the shift count is loaded into the Shift Control Counter, SC, and then a loop is executed. During every iteration of the shift loop a signal (SHLA, SHLAQ, SHRA, or SHRAQ) is activated which causes a one-bit shift to be accomplished by the SHIFTL or SHIFTR program. Every iteration of the loop also activates a variable, STEPSC, which decrements the bit count in SC. The shift loop terminates when SC equals zero. Thus, the SC register controls the number of positions which the accumulator is shifted.

The Shift Control Counter program (SCCTR) indicates how the STEPSC signal is used to decrement the SC register. Since this program is particularly simple and straightforward, it will be used to illustrate the process of automatic design. This program is shown in Figure 3 as it appears in the 1800 description.

The "MM" heading in the first card of this program identifies it as a Micro-program with Manually specified control. This prevents ALERT from generating sequence control logic. Instead, this section depends for control only upon the signals explicitly designated in the program.

The program consists of just two assignment statements. The first statement assigns signals to the set side of the SC register flip-flops, and the second assigns signals to the reset side. These signals, which are Boolean functions of the current state of the SC bits, are inhibited except when STEPSC = 1. Since ALERT provides no extra control logic in this program, an assignment will occur whenever STEPSC is pulsed. At such an occurrence SC will acquire a new value, and this value — as the reader may verify — will constitute a decrease by one of the binary number represented in SC previously.

When the input deck was processed by the ALERT processor, the Shift Control Counter program was translated into the logic equations shown in the print-out in Figure 4. The original assignment statements were decomposed into elementary statements, each of which defines a single gating function and identifies signal sources and destinations. For example, the first equation specifies that the complement of bits 0, 1, 2, 3, 4, and 5 of SC are ANDed together and the resultant variable is designated as "T59." Such "T" variables are intermediate elements supplied automatically during translation of the source statements. This term corresponds to the sub-expression,

" ('AND'/'NOT'SC) ",

in the first assignment statement of the source document (Figure 3). During further processing of the 1800 design, variables representing signals from other sections of the design were combined with equations in the SCCTR section. The resulting equations indicate need of 14 two-input AND gates, two 3-input ANDs, two 4-input ANDs, two 5-input ANDs, two 6-input ANDs, 12 one-input OR gates, and 12 two-input OR gates. Thus a total of 46 elementary logic gates, or "blocks," are required.

When these equations were processed by the IBM Logic Automation programs, the 12 one-input OR gates were discovered to be redundant, and the revised block count dropped to 34.

The designs were processed further in order to comply with engineering and circuitry constraints of Solid Logic Technology. Extra gating was automatically provided and the block count then rose to 102. Six pages of logic schematics were automatically drawn from the resultant design of the SCCTR section. The first of these pages is shown and discussed in Appendix III.

The design of the SCCTR section as generated by ALERT is shown in summary form in Figure 5. The design of the corresponding section of the actual machine is represented in Figure 6. For simplicity, in both these figures special gating imposed by the circuit connection rules has been omitted, and signals and gates not related to the shift counter decrement process also are not shown. In the real 1800 the decrement signal (here called "DESC") is gated only to the least significant bit (SC/1) which will toggle when a pulse is applied (see Figure 6).* If SC/1 is in the "0" state, changing it to "1" will cause toggling of the next higher bit, SC/2. Likewise every SC bit will, when toggled, itself toggle the next higher bit. This chain reaction may propagate to the highest bit of the counter, but will cease at a bit with an original value of 1. This process causes the value represented in SC to be decremented.

The design generated by ALERT employs a different method of counting. This difference, however, does not result from the ALERT system itself but is due to the way the counting function had originally been specified in the APL description. In the design produced automatically, the counter is organized so that the decrement signal is applied to all six bits of SC simultaneously rather than by a bit-by-bit propagation mechanism.

The logic generated by ALERT used AND gates having from two to seven inputs each (Figure 5). However, after automatic revision to meet circuit needs, there was an increase in gates required. For example, in terms of the idealized logic generated by ALERT, two gates were used to provide the signal to set the most significant bit of SC

* Subscripting conventions differed in the output generated by ALERT and in the engineering documents of the real machine. In the ALERT output, the six bits of SC are denoted, from most significant to least, as SC/0, SC/1, ..., SC/5. In the real 1800, these bits are denoted, respectively, as SC/32, SC/16, ..., SC/1.

(Figure 7(a)). When this design was converted into circuitry, 13 gates were required, as shown in Figure 7(b). This section of design is also shown in final printed form in Appendix III. This discrepancy between the automatically produced design and the manual design is due mainly to restrictions on the available circuit types. Because inversion takes place for all circuits used, to get the proper polarity, extra logic levels were needed. This effect is aggravated by fan-in restrictions and circuit card layout considerations. Also, because of the partitioning of design, signals generated in one section may not be available in another section. This is the case for $\overline{SC/0}$ and $\overline{SC/1}$. Inverters are used to generate these complement signals from SC/0 and SC/1 instead of obtaining them directly from another section. For the entire shift counter, a total of 102 logic elements are generated by ALERT as against 36 for the real 1800. If the projected improvements are made in ALERT, the count of the logic blocks generated is expected to be reduced to 42.

```
** SCCTR.       SHIFT COUNTER DECREMENT FACILITY                                  SCCTR 0
                                                                                  SCCTR 01
     SC =   'SET' STEPSC 'AND' ('AND'/'NOT'SC),                                   SCCTR 02
                               ('AND'/'NOT'SUFF(X(5)/SC),                         SCCTR 04
                               ('AND'/'NOT'SUFF(X(4)/SC),                         SCCTR 06
                               ('AND'/'NOT'SUFF(X(3)/SC),                         SCCTR 08
                               ('AND'/'NOT'SUFF(X(2)/SC),                         SCCTR 10
                                'NOT'SC(5).                                       SCCTR 12
                                                                                  SCCTR 13
     SC = 'RESET' STEPSC 'AND' ('AND'/SC(0),'NOT'SUFF(X(5)/SC),                   SCCTR 14
                               ('AND'/SC(1),'NOT'SUFF(X(4)/SC),                   SCCTR 16
                               ('AND'/SC(2),'NOT'SUFF(X(3)/SC),                   SCCTR 18
                               ('AND'/SC(3),'NOT'SUFF(X(2)/SC),                   SCCTR 20
                               ('AND'/SC(4),'NOT'SC(5)),                          SCCTR 22
                                SC(5).                                            SCCTR 24
```

Fig. 3. The Shift Control Counter program.

```
EQUATIONS
T59=-SC/0*-SC/1*-SC/2*-SC/3*-SC/4*-SC/5
T60=-SC/1*-SC/2*-SC/3*-SC/4*-SC/5
T61=-SC/2*-SC/3*-SC/4*-SC/5
T62=-SC/3*-SC/4*-SC/5
T63=-SC/4*-SC/5
T64/0=STEPSC*T59
T64/1=STEPSC*T60
T64/2=STEPSC*T61
T64/3=STEPSC*T62
T64/4=STEPSC*T63
T64/5=STEPSC*-SC/5
T514/0=T64/0+T112/0
T514/1=T64/1+T112/1
T514/2=T64/2+T112/2
T514/3=T64/3+T112/3
T514/4=T64/4+T112/4
T514/5=T64/5+T112/5
SC/S/0=T514/0
SC/S/1=T514/1
SC/S/2=T514/2
SC/S/3=T514/3
SC/S/4=T514/4
SC/S/5=T514/5
T65=SC/0*-SC/1*-SC/2*-SC/3*-SC/4*-SC/5
T66=SC/1*-SC/2*-SC/3*-SC/4*-SC/5
T67=SC/2*-SC/3*-SC/4*-SC/5
T68=SC/3*-SC/4*-SC/5
T69=SC/4*-SC/5
T70/0=STEPSC*T65
T70/1=STEPSC*T66
T70/2=STEPSC*T67
T70/3=STEPSC*T68
T70/4=STEPSC*T69
T70/5=STEPSC*SC/5
T515/0=T70/0+T117/0
T515/1=T70/1+T117/1
T515/2=T70/2+T117/2
T515/3=T70/3+T117/3
T515/4=T70/4+T117/4
T515/5=T70/5+T117/5
SC/R/0=T515/0
SC/R/1=T515/1
SC/R/2=T515/2
SC/R/3=T515/3
SC/R/4=T515/4
SC/R/5=T515/5
END OF EQUATIONS
```

Fig. 4. Print-out of Boolean equations generated for the Shift Control Counter.

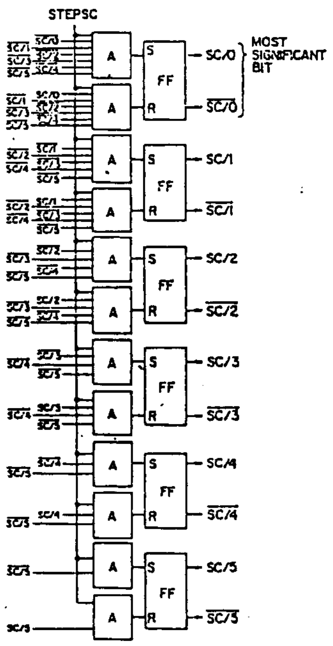


Fig. 5: Summary of the automatically produced design of the Shift Control Counter.

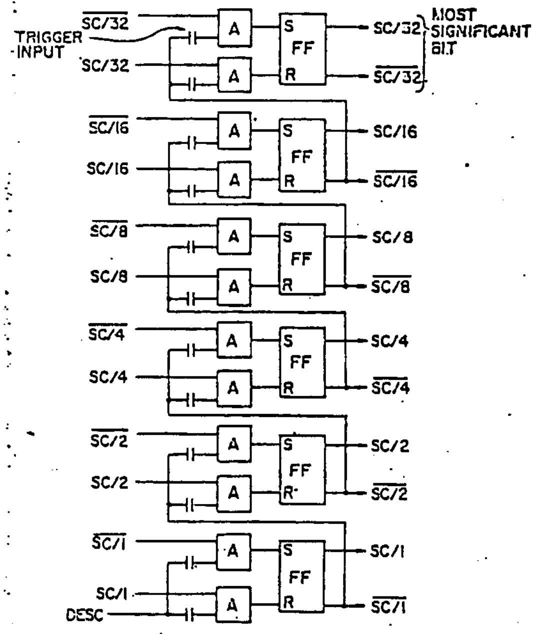


Fig. 6. Summary of the actual design of the Shift Control Counter.

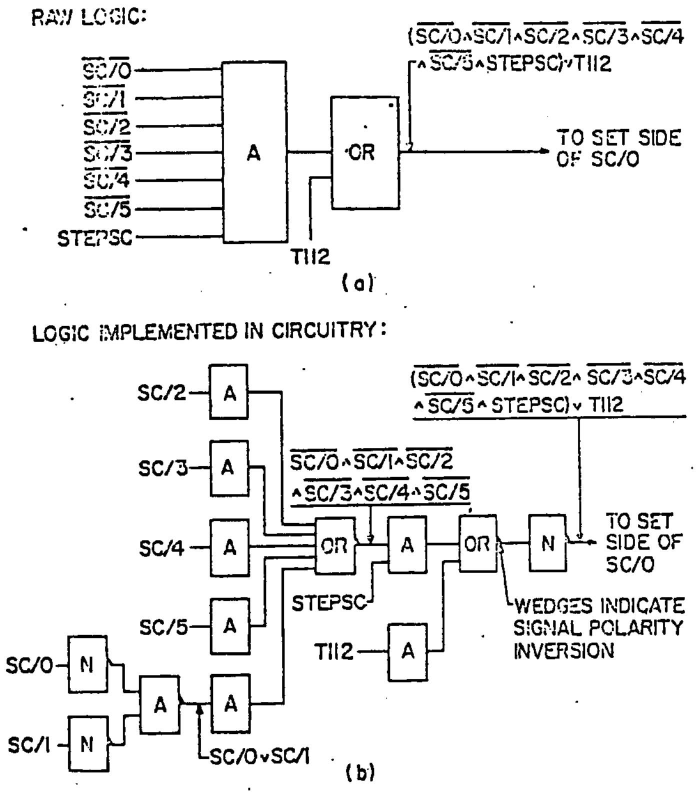

RAW LOGIC:
$\overline{SC/0}$
$\overline{SC/1}$
$\overline{SC/2}$
$\overline{SC/3}$
$\overline{SC/4}$
$\overline{SC/5}$
STEPSC
A
OR
TII2
$(\overline{SC/0} \wedge \overline{SC/1} \wedge \overline{SC/2} \wedge \overline{SC/3} \wedge \overline{SC/4} \wedge \overline{SC/5} \wedge STEPSC) \vee TII2$
TO SET SIDE OF SC/0
(a)
LOGIC IMPLEMENTED IN CIRCUITRY:
SC/2
SC/3
SC/4
SC/5
SC/0
SC/1
N
A
OR
STEPSC
TII2
$\overline{SC/0} \wedge \overline{SC/1} \wedge \overline{SC/2} \wedge \overline{SC/3} \wedge \overline{SC/4} \wedge \overline{SC/5}$
$(\overline{SC/0} \wedge \overline{SC/1} \wedge \overline{SC/2} \wedge \overline{SC/3} \wedge \overline{SC/4} \wedge \overline{SC/5} \wedge STEPSC) \vee TII2$
TO SET SIDE OF SC/0
WEDGES INDICATE SIGNAL POLARITY INVERSION
SC/0 v SC/1
(b)

Fig. 7. Idealized logic and its implementation in circuit blocks.

## APPENDIX III

Six pages of schematic [illegible] diagrams were produced automatically from the designs of the Shift Control Counter section. The first diagram is shown in Figure 8. This diagram represents the logic to provide the signal, SC/S/0, to set bit 0 of the SC register.

A large proportion of the resulting gates serve only to satisfy circuitry requirements rather than to carry out logical functions. It is not possible to investigate the quality of design without consideration of circuit connection rules. But the actual rules differ from one family of circuitry to another and are not part of the logic generation processes of ALERT. Although circuit connection problems are mentioned in this section, these topics cannot be adequately covered here.

At the lower left, two N blocks (single-input inverters) are used to generate the $\overline{SC/0}$ and $\overline{SC/1}$ signals from SC/0 and SC/1. (Due to formatting conventions, the slash in these names is not printed on the drawings.). Since $\overline{SC/0}$ and $\overline{SC/1}$ are also needed in other drawings, two lines are shown tapping off from them. $\overline{SC/0}$ and $\overline{SC/1}$ are used as inputs to an AND inverter, the output of which provides the function SC/0 ∨ SC/1. This gate is used to reduce the number of input lines to the OR inverter at the center of the drawing. Five single-input AND gates feed this OR inverter. (These ANDs do not serve a logic function but are required for circuit needs.) The inputs to the five AND gates are SC/2, SC/3, SC/4, SC/5 and the combined signal SC/0 ∨ SC/1. Thus the output of the OR inverter becomes $\overline{SC/0} \wedge \overline{SC/1} \wedge \overline{SC/2} \wedge \overline{SC/3} \wedge \overline{SC/4} \wedge \overline{SC/5}$. This output is then ANDed with the stepping pulse STEPSC, and the result defines the set side of SC/0. However, another signal, T112, not related to counting, is also needed as a set input. To enable them to use a single line to SC/0, the stepping control and the T112 signals are ORed together. Since the circuit used for the OR function also inverts the output, an extra N inverter is used to restore it to correct polarity. The final result (SC/S/0) is thus

$$(\overline{SC/0} \wedge \overline{SC/1} \wedge \overline{SC/2} \wedge \overline{SC/3} \wedge \overline{SC/4} \wedge \overline{SC/5} \wedge \text{STEPSC}) \vee \text{T112}.$$

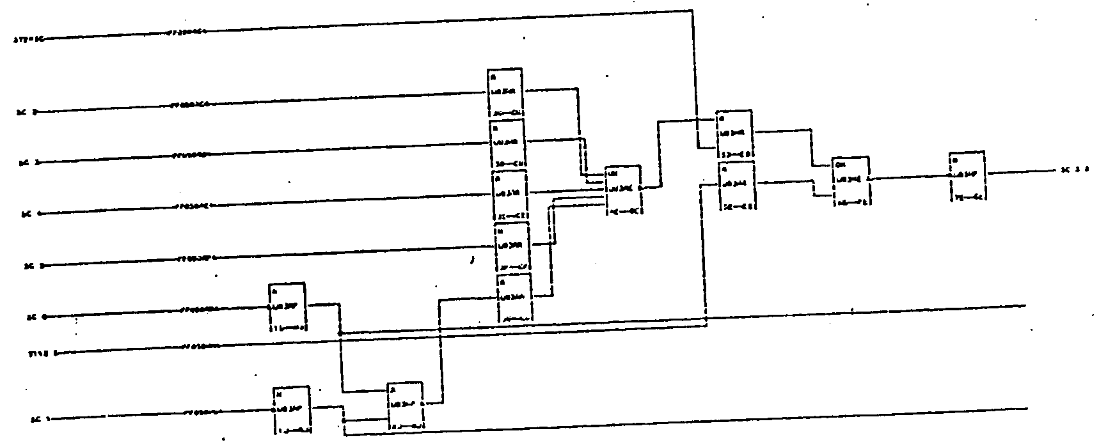


Fig. 8. A schematic logic diagram produced automatically from output from ALERT

## Acknowledg[illegible]ment

[illegible] basic approaches used in this sys[illegible] [illegible]ally proposed by C. [illegible] Coleman, [illegible] [illegible] Angeles Scientific Center.

## References

1. [illegible] A. Breuer, "General survey of design auto-[illegible]tion of digital computers," Proceedings of [illegible] IEEE, Vol. 54, pp. 1708-1721, December [illegible]66.

2. [illegible] Burgbacher, "Proce[illegible]or-controller of IBM [illegible]0 system, a formal description," IBM Tech-[illegible]cal Note TN 02.559.[illegible], July 1966.

3. [illegible] W. Case, H. H. Graff, L. E. Griffith, A. R. [illegible]Clercq, W. B. Murley, and T. M. Spence, [illegible]lid logic design automation for IBM Sys-[illegible]/360," IBM J. Res. and Dev., Vol. 8, pp. [illegible]-140, April 1964.

4. [illegible] Chu, "An ALGOL-like computer design langu-[illegible]ge," Communications of the ACM, Vol. 8, pp. [illegible]07-615, October 1965.

5. [illegible] Estrin and R. Mandell, "Metacompiler as a design automation tool," 1966 Proc. Share De-[illegible]gn Automation Conference.

6. D. Falkoff, K. E. I[illegible]erson and E. H. Sussen-[illegible]th, "Formal description of System/360," IBM Sys. J., Vol. 3, No. [illegible], pp. 198-262, 1964.

7. T. D. Friedman and S. C. Yang, "Methods used in an automatic logic design generator (ALERT)," IEEE Trans. on Computers, Vol. C-18, No. 7, pp. 593-614, July 1969.

8. ________, ALERT: A program to compile de-[illegible] of new computers," Digest 1st Annual [illegible] Computer Conference, pp. 128-130, Sep-[illegible]ber 1967.

9. [illegible] F. Gorman and J. P. Anderson, "A logic de-[illegible]n translator," 1962 Proc. FJCC, pp. 86-96.

10. [illegible] E. Iverson, A Programming Language, New [illegible]: Wiley, 1962.

11. [illegible] Metze and S. Seshu, "Computer compiler [illegible] I — Perliminary report," Coordinated [illegible]ence Lab., University of Illinois, Urbana, [illegible]. R-303, August 1965.

12. [illegible] Proctor, "A logic design translator exper-[illegible] demonstrating relationship of language [illegible]ystems and logic design," IEEE Trans. on [illegible]ctronic Computers, Vol. EC-13, pp. 422-430, [illegible] 1964.

13. [illegible] Roth, "Systematic design of automata," [illegible] Proc. FJCC, pp. 1093-1099.

14. [illegible] Schlaeppi, "A formal language for de-[illegible]ing machine logic, timing and sequencing [illegible])," IEEE Trans. on Electronic Computers, Vol. EC-13, pp. 439-448, August 1964.

15. "[illegible] 1800 Functional Characteristics," IBM Sys-[illegible] Reference Library, Form A26-5918.

16. J. R. Duley and D. L. Dietmeyer, "Translation of a DDL Digital System Specification to Boolean Equations," IEEE Trans. on Computers, Vol. C-18, No. 4, pp. 305-313, April 1969.

12

#71

AR-6

17th DA Conference 1980

VINCENT N. TRANS
EXAMINER
ART UNIT 234

# A New Look at Logic Synthesis

John A. Darringer
William H. Joyner, Jr.

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

## Abstract

Despite many attempts to generate hardware implementations automatically from functional specifications, the literature does not record any commercial success. Previous efforts have dealt primarily with technology-independent primitives and have emphasized circuit minimization. However, larger scales of integration have made other design requirements and technology restrictions as important as circuit count, and have increased the cost of making an engineering change. Thus it is becoming increasingly important to insure that initial chip designs are correct. This paper outlines an investigation into the feasibility of logic synthesis in this new context. A system is described which will produce a naive implementation automatically from a functional specification, and then will interact with the designer, allowing him to evaluate it with respect to these many factors, and to improve it incrementally by applying local transformations until it is acceptable for manufacture. The use of simple local transformations will insure correct implementations, will isolate technology-specific data, and will allow the total process to be applied to larger VLSI designs. This approach has been tested on the design of a single chip with encouraging results. A prototype synthesis system is now being used to perform further experiments.

## Logic Synthesis

The goal of logic synthesis is to accept functional specifications for a hardware unit and to generate automatically a detailed, technology-specific implementation comparable in quality to that of an experienced engineer. The nature of this problem depends on the level of the functional description, the set of implementation primitives, and the criteria of acceptability. Initially, we are examining a restricted form of synthesis. That is, we are concentrating on the problem of generating masterslice implementations from low-level register transfer specifications in which all memory elements are specified. The combinational network between the memory elements must be generated and the entire function must be implemented with primitives from a specified set. Further, the implementation must satisfy given performance requirements and technology restrictions. We hope to determine the feasibility of interactive synthesis in this context, and the applicability of techniques of program analysis and optimization.

There has been much work on automating logic design and, although many effective tools have been developed to aid the implementer, synthesis is not considered commercially successful. Early work centered on developing algorithms for translating a boolean function into a minimum two-level network of boolean primitives. Extensions were developed for handling limited circuit fan-in and alternative cost functions [Bre72, Die71]. Because these algorithms search for minimal implementations they are necessarily exponential and require too much space and time to be used on most actual designs.

Later efforts attempted to raise the level of specification. The DDL work at Wisconsin [Dul68, Die71], APDL at Carnegie-Mellon University [Dar69], and ALERT at IBM [Fri68] all began with behavioral specifications and produced technology-independent implementations in the form of boolean equations. The results were usually more expensive than manual implementations and did not take advantage of the target technology. For example, the ALERT system was validated on an existing design, the IBM 1800, and the implementation produced required 160% more circuits than the manual design [Fri70].

In attempts to generate more efficient logic and to give the user more control over the implementation, other strategies were tried [Sch62, Mes68, Hil73]. These constrain the specification language so that there is nearly a one-to-one correspondence between the specification and the implementation. Of course, this constraint also decreases the advantage provided by the system.

The RT-CAD project at Carnegie-Mellon [Sie76] has developed tools to support the total machine design process from architectural design to physical layout. So far, some tools have been demonstrated for the early part of the design cycle, but the total system is not yet complete [Bar73, Tho77, Sno78, Haf78]. There is also an effort in Japan to develop a system to help a designer translate an existing SSI or MSI implementation into LSI [Nak78].

© 1980 ACM [illegible] $0.75

## A New Approach

The system we propose is not intended to be a completely automatic replacement for the manual design process. Instead, it is an interactive system in which the user operates on a logic design at several levels of abstraction. He can simplify an implementation at one level, and when satisfied can move to the next level. He does this by applying transformations, either locally or globally, to achieve the simplification or refinement. By being able to operate on the implementation at several levels, the user can often make a change at one level that will cause a great simplification at a lower level.

Our approach differs from previous efforts in other respects. We see the logic synthesis problem as one of finding a feasible (not an optimal) implementation: a network of primitive boxes that satisfies a large number of constraints. These include timing constraints, a restricted library of primitives, driver requirements, clock distribution rules, fan-in and fan-out constraints, rules for testability (LSSD), and switching currents, in addition to limits on gate count and I/O pins. Though previous attempts have considered some of these factors, most have been limited to technology-independent transformations and have not given full consideration to all the requirements and restrictions placed on the logic designer today.

We are also attempting to limit our transformations to local changes that do not require exponential time or space. If we are successful then our approach may scale up to handle larger VLSI chips. Previous efforts have relied on algorithms that do not scale to realistic problems.

In designing a system to apply these transformations, we have been influenced by the design of optimizing compilers that translate a higher level description into a low level implementation through several intermediate levels, which are convenient for analysis and optimization. Similarly, we first convert a specification in a hardware description language to a network of typed boxes: ANDs, ORs, NOTs, memories, parity checkers, etc. We then allow transforms to be applied to this network to move the design closer to one that can be implemented in the available primitives of the target technology.

Our initial task is to identify the factors that influence a designer, to develop local transformations for altering these factors, and to learn how to apply these transforms (globally or at specific points) to generate implementations in an actual target technology. If these efforts are successful we could expect shorter hardware development times, lower development costs, and fewer errors since we will show that the transformations preserve the specified function.

## A System for Logic Synthesis

The principal input to a synthesis system is a functional description of the logic to be implemented. We accept this description in a language similar in level to CDL [Chu65], and transform it into a network of boolean functions, memories, inputs, and outputs in a straightforward manner. Initially, we are considering functions implemented by a single chip. Therefore, we require an interface description giving the polarities of the inputs available and the outputs required, and the sources and destinations of inputs and outputs -- information needed to assign receivers and drivers. Information is also needed about the target technology and its design rules, describing the primitives available, the fan-in and fan-out requirements, and the timing constraints. This information is held in a technology file.

The analysis involved in logic synthesis takes place at several levels of abstraction. Transformations such as identification of common sub-expressions will be appropriate at every level, while simultaneous switching could be more conveniently analyzed at a high level (where registers and decoders are easily recognizable) and timing analysis at a lower level (where the real hardware primitives are available). At each level the description will be in terms of primitives of that level, such as registers, shifters, and decoders or at a lower level shift register latches, and-inverts, and dot-ands. These primitives will have two interpretations: a set of properties convenient for analysis, and an implementation (possibly parametric) in terms of lower level primitives. This is in contrast to a "macro" approach where the only meaning of a box is its expansion in terms on the lowest level primitives.

Figure 1 shows the organization of the logic synthesis system. The inner box delimits the concern of this paper. Its inputs are the register transfer specification, the interface constraints, and the design rules which apply to the target technology. The output is a detailed implementation in terms of the primitives of the target technology, which is submitted to placement and wiring. Though gate and pin restrictions and rough timing constraints should have been enforced by the transforms, some violations of these may be apparent only after placement and wiring are complete. The loop back into the synthesis system permits other attempts at synthesis, either with adjusted design rules and technology parameters or with different user input, until a wirable implementation is achieved.

## An Experiment

To investigate the feasibility of the approach to synthesis outlined above, we conducted an experiment with a chip of an existing processor. This chip had been specified at the functional level and then implemented by engineers. The existence of the engineers' implementation of the chip allowed us to compare the two implementations and to study the differences. The transforms of the experiment were performed initially by hand and later by our prototype system, as a first step in learning how logic design in this context is done and what parts of it are amenable to automatic methods.

The initial description used as input to this experiment was similar to a CDL [Chu65] description. It specified the function that was to be implemented by a single chip, the function's inputs and outputs, and, importantly, each memory element that is to appear in the final implementation. For generating a technology-specific implementation, moreover, additional information about the inputs and outputs will be needed: polarity of inputs available, polarity of outputs re-

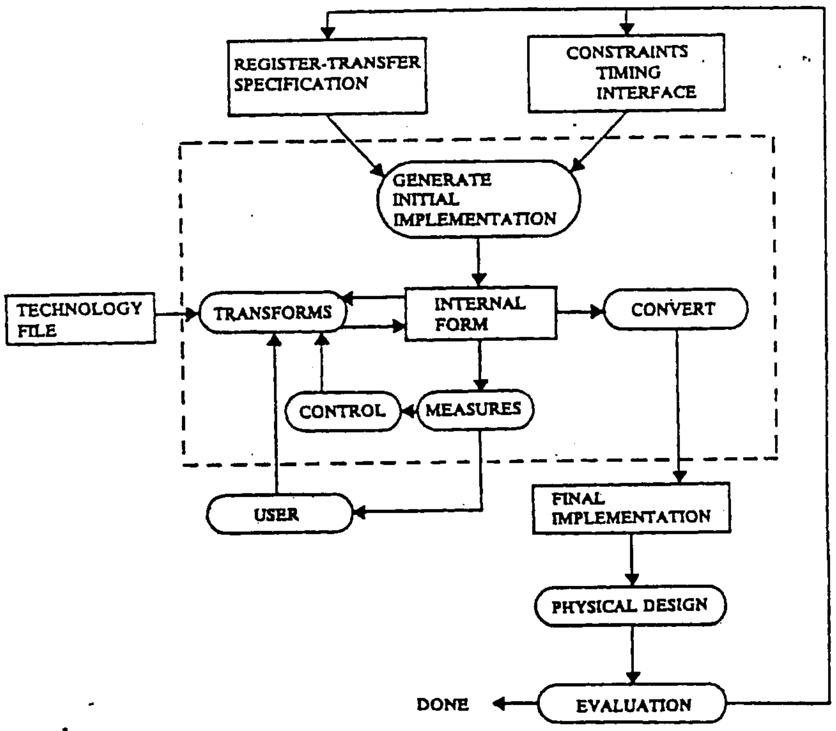


Figure 1. The Logic Synthesis System

quired, source of inputs and destination of outputs (for receiver and driver requirements), and frequency of change of inputs (to determine timing requirements). This information is part of an interface specification separate from the functional description and used later in the sequence of transformations.

Also required for the synthesis system is information about the target technology, either in a technology file or built into the transformations. In this experiment a logic chip is a masterslice design with 96 I/O pins and 704 sites, equally divided between 3- and 4-input NAND gates. The designer, however, also uses macro-like definitions of higher level boxes made up of these NAND gates. In addition, the technology file contains restrictions on the use of the primitives and macros, fan-in and fan-out requirements, timing constraints, and clocking and powering rules.

An important part of our approach is an "internal form" that is capable of representing the implementation at different levels of abstraction. Our system to support logic synthesis makes use of a graph-like internal data structure for storing the implementation as it progresses from the higher-level description to its final form, and transformations operate on this graph. All descriptions are synchronous models of hardware: there is an implicit clock and on each 'tick' the state of the machine is defined. The computation of the next state and outputs in terms of the current state and inputs proceeds between the ticks. There is a single organizational component: the 'box'. A box has input and output terminals which are connected by wires to other boxes. Each box also has a type, which may be primitive or may reference a definition in terms of other boxes. Thus a hierarchy of boxes can be used, and an instance of a high-level box such as a parity box can be treated as a single box or expanded into its next level implementation when that is desirable.

**The Initial Implementation**

The first step is to translate the functional specification into a network of high-level functions to form an initial, naive implementation. The difficulty of this task depends on the level of the functional specification. In our case local implementations of each specification language construct are substituted in a straightforward fashion. While the resulting initial implementation is in some cases extremely naive, the generator of this implementation is simple, which reduces the task of insuring that it produces correct implementations.

**The Transformations**

A library of transformations, which can be applied to the initial implementation or any implementation, is available. Not all transforms are applicable at every level, and there is some choice in selecting a particular application sequence. It is here that the interaction with the user is most important. Figure 2 summarizes the sequence of transforms used in this experiment; the application sequence might be very different in other experiments. Our hope is that after further experimentation we will find a set of transformations and an application sequence that produce acceptable implementations for a large class of chip specification with little or no user intervention.

We have made some decisions in ordering of the transforms. For example, there are three simplification steps shown in Figure 2. One might question whether simplification both before and after the generation of NANDs is necessary. The inclusion of both steps reflects a tradeoff between applying transforms as early as possible to reduce the size of the network to which later transforms are applied, and having fewer transforms. Another tradeoff exists in the writing of transforms themselves. For reasons of efficiency one might write one transform so that it called another after making a change which suggested that the other transform would apply. On the other hand, this makes the individual transforms more complicated and less flexible.

The synthesis process is an alternating sequence of descriptions and transformations. These intermediate descriptions and the transforms used to produce them are summarized below:

Level 1. The starting point is a naive implementation generated from the functional description in a straightforward fashion. In our example this is a network of 183 boxes (e.g. AND, OR, REGISTER, and PARITY). Box count is only one measure and is listed here merely to give the reader a feel for the effect of the various transforms.

Level 2. Here simple local transformations are applied to the initial implementation and substantially reduce the number of boxes. Examples of the transforms used are:

$$\mathrm{AND}(a,\mathrm{NOT}(a)) \rightarrow 0$$
$$\mathrm{OR}(a,\mathrm{NOT}(a)) \rightarrow 1$$
$$\mathrm{OR}(a,\mathrm{AND}(\mathrm{NOT}(a),b) \rightarrow \mathrm{OR}(a,b)$$
$$\mathrm{XOR}(\mathrm{PARITY}(a_1,\ldots,a_n),b) \rightarrow \mathrm{PARITY}(a_1,\ldots,a_n,b)$$

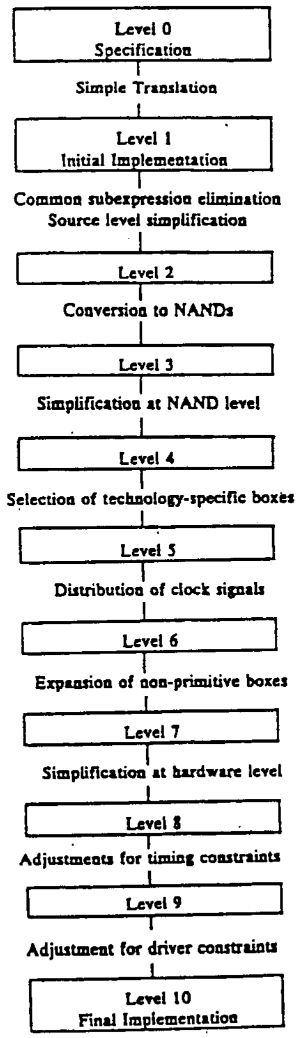

Level 0
Specification
Simple Translation
Level 1
Initial Implementation
Common subexpression elimination
Source level simplification
Level 2
Conversion to NANDs
Level 3
Simplification at NAND level
Level 4
Selection of technology-specific boxes
Level 5
Distribution of clock signals
Level 6
Expansion of non-primitive boxes
Level 7
Simplification at hardware level
Level 8
Adjustments for timing constraints
Level 9
Adjustment for driver constraints
Level 10
Final Implementation

Figure 2 The steps of the synthesis experiment

These transformations, as well as the initial generation, may introduce constant inputs 0 and 1 for boxes, and may generate several copies of a box. Transformations are applied to eliminate these through constant propagation and common subexpression elimination, techniques borrowed from optimizing compilers. These changes, in turn, may produce boxes which have either no inputs or no outputs. These are successively removed in a technique analogous to unreachable code elimination. As a result of these steps the implementation is reduced to 83 boxes.

Level 3. The AND, OR, NOT, and most other operators of the initial description are replaced by NAND implementations. These transformations are local and may introduce unnecessary double NANDS that will be eliminated later. Also idealized senders and receivers are installed at the chip interface. Some higher-level operators such as EQ and PARITY are not replaced now but will be expanded later. At this point the implementation contains 174 boxes.

Level 4. Simplifying transforms are applied to each signal in the network. These transforms are local in that they replace a small subgraph of the network (usually five boxes or fewer) by another subgraph which is functionally equivalent but simpler according to some measure. They are used throughout the network until no more apply. Elimination of "double negation", a sequence of two single-input NANDs, is the transform most used at this step. This and another gate-reducing transform are shown in Figure 3.

When the nand transforms are applied globally the reduce the number of boxes to 158. However, a designer would notice that one signal network that is still overly complex. On close examination he would find that the fan-out of intermediate signals prevents the transforms from applying. He can intervene at this point, eliminate the fan-out by introducing duplicate logic, and reapply the transforms to further reduce box count and path length. Final box count is 153.

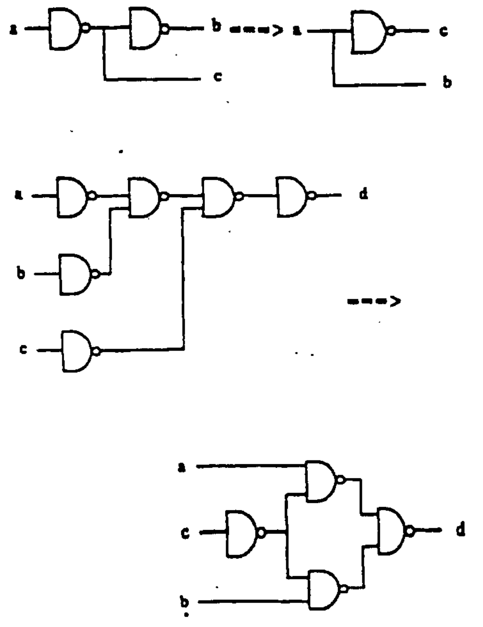


Figure 3 Examples of NAND Transformations

Level 5. At this level the first technology-specific transforms are applied. Specific networks of gates in the target technology are selected to implement the NANDs, idealized registers, drivers, and other boxes present in the previous level. Some manipulation may be necessary to match the fan-in of the actual primitives with that of the "idealized" boxes. Also the number of control and data lines of the idealized registers might exceed those actually available, and necessitate the generation of additional logic. The logic at this level is in terms of primitives used by the engineers in their implementations, but may not obey all timing, fan-out, and other technology restrictions. In some cases this is a many-to-one mapping, and the resulting size is 93 boxes.

Level 6. Up to this point the latches are assumed to be clocked by an implicit clock. Now that actual latches have replaced the idealized registers, actual clock signals must be attached. The technology file contains the rules for distributing the clock to the latches. The implementation now contains 134 boxes.

Level 7. At this point the higher-level operators such as PARITY are expanded in terms of the hardware primitives. The definitions of these boxes will have been synthesized themselves, either by hand or with automatic aid. For example, the two input EQ function can be replaced by a network of three NAND gates realized in the technology. After expansion there are 192 boxes.

Level 8. Replacement of boxes by lower level primitives may make further gate-saving simplifications possible, especially at the "borders" of the expansions and the previously existing logic. Technology-specific analogs of some of the transforms of Level 5 may be applied. Also, wired (dot) ANDs may be introduced if allowed by the technology; they may be used to eliminate gates if the inputs to a NAND-realized AND are not used elsewhere. The simplified implementation contains 159 boxes.

Level 9. The technology may specify a maximum and minimum number of gate delays allowed between latches. Though complete timing analysis may not be possible until after placement and wiring, some guidelines can be enforced by transforms at this level, by adding gates to short paths or by reducing numbers of levels to shorten long paths. Information about whether input signals change rapidly enough to make their delays critical, supplied with the chip interface information, may be used to avoid unnecessary delays. After this adjustment there are 188 boxes.

Level 10. The fan-out of each signal is adjusted to conform to the technology constraints, represented by inequalities in the technology file. Power requirements may require the source box of a signal with fan-out exceeding these limits to be duplicated to reduce this fan-out. In our experiment the box count remained at 188. While the 188 generated boxes are primitives for designers, they are in fact macros requiring several master-slice gates for implementation. The fully expanded implementation nearly fills the chip.

DEF012028

## Results

The synthesized chip was compared with the existing implementation, and no significant differences were detected. In fact the synthesized implementation used five fewer gates than that of the engineer, and satisfied all technology constraints. An example of the differences was that two unnecessary delays included by the engineer were omitted in our synthesis. Another difference was in the use of dot ANDs. Considering that the experiment involved a single chip, that we had access to the manual implementation, and that we selected the order of application of transforms, it should not be a surprise that the two implementations were in such close agreement.

## Future

During the coming year we plan to continue the development of the prototype synthesis system and to use it in conducting further experiments. We plan to look at more ambitious chips -- chips that have required minimization or that have caused long path problems, when implemented manually. We may find that additional user direction will be necessary. This may require additional measures and transforms, but our hope is that new chips will not always require additional transforms and that we can begin to develop sequences of transforms and higher-level transforms that will make the total task more automatic. In addition, we will explore:

- multi-chip synthesis -- starting with a functional specification that requires several chips, developing additional measures and transforms that will trade resources across chip boundaries.

- alternative technologies -- looking at the sensitivity of synthesis to alternative technologies and determining how the synthesis system could respond to changing technology.

- engineering changes -- examining how such a synthesis system could respond to EC's where minimum, local changes are highly desirable.

- transform specification -- looking at how transforms could be described at a high level and compiled for efficient application.

- transform correctness -- considering what properties should be proved of transforms such as function preserving or power decreasing, and demonstrating how such proofs can be accomplished.

## Summary

We are in the midst of taking what we believe is a new approach to the old problem of logic synthesis and are encouraged by our preliminary experiments. We have built a prototype synthesis system that allow us to perform more ambitious experiments to validate our approach. Our hope is that computationally manageable techniques based on local transformations can be applied to improve naive implementations to acceptable ones. This could greatly shorten processor development and validation times.

## Acknowledgements

We would like to thank William van Loo for many helpful discussions on master-slice chip design. Also Leonard Berman, Louise Trevillian, and Thomas Wanuga have made valuable contributions to the design and implementation of the prototype synthesis system.

## References

[Bar73] Barbacci M, "Automated Exploration of the Design Space for Register Transfer Systems", PhD Thesis, CS Dept., Carnegie-Mellon University, 1973.

[Bre72] Breuer M A (Editor), *Design Automation of Digital Systems*, Prentice-Hall, 1972.

[Chu65] Chu Y, "An ALGOL-like Computer Design Language", *Communications ACM*, Oct 1965.

[Dar69] Darringer J A, "The Description, Simulation, and Automatic Implementation of Digital Computer Processors", Ph.D. Thesis Carnegie-Mellon University 1969.

[Die71] Dietmeyer D L, *Logic Design of Digital Systems*, Allyn and Bacon, 1971,1978.

[Dul68] Duley J R, "DDL -- A Digital Design Language", Ph.D. Thesis, University of Wisconsin 1968.

[Dul69] Duley J R and D L Dietmeyer, "Translation of a DDL Digital System Specification to Boolean Equations", *IEEE TC Vol C-18*, April 1969.

[Fri68] Friedman T D and S C Yang, "Methods used in an Automatic Logic Design Generator(ALERT)," *IEEE TC Vol C-18*, July 1968.

[Fri70] Friedman T D and S C Yang, "Quality of Design From an Automatic Logic Logic Generator(ALERT)," Proc. 1970 Design Automation Conference.

[Haf78] Hafer L J and A C Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process", Proc. 15th Design Automation Conf., 1978.

[Hil73] Hill F J and G R Peterson, *Digital Systems: Hardware Organization and Control*, Wiley, 1973.

[Mes68] Mesztenyi C K, "Computer Design Language Simulation and Boolean Translation", TR68-72, CS Dept., Univ. of Maryland, 1968.

[Nak78] Nakamura S et.al., "LORES - Logic Reorganization System", Proc. 15th Design Automation Conf., 1978.

[Sch62] Schorr H. "Toward the Automatic Analysis and Synthesis of Digital Systems" Ph.D. Thesis, Princeton, 1962.

[Sie76] Siewiorek D P and M R Barbacci, "The CMU RT-CAD System -- An Innovative Approach to Computer Aided Design", Proc. AFIPS Conf. Vol. 45, 1976.

[Sno78] Snow E A, "Automation of Module Set Independent Register-Transfer Level Design", PhD Thesis, EE Dept., Carnegie-Mellon University, 1978.

[Tho77] Thomas D E, "The Design and Analysis of an Automated Design Style Selector", PhD Thesis, EE Dept., Carnegie-Mellon University, 1977.

13

AS-6

#75

# Experiments in Logic Synthesis

John A. Darringer
William H. Joyner
Leonard Berman
Louise Trevillyan

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

## Abstract

The rise in circuit density and processor complexity is driving up the cost of engineering changes and making it increasingly important to insure that initial chip implementations are correct. No longer is the logic designer concerned solely with minimizing circuit count. Instead, he is faced with a growing number of design requirements and technology restrictions. This paper describes experiments with an automated system that allows a designer to start with a naive implementation produced automatically from a functional specification, evaluate it with respect to these many factors, and incrementally improve this implementation by applying local transformations until it is acceptable for manufacture. The use of simple local transformations in this system insures correct implementations, isolates technology-specific data, and allows the total process to be applied to larger, VLSI designs.

## Background

The goal of logic synthesis is to accept functional specifications for a hardware unit and to generate automatically a detailed, technology-specific implementation comparable in quality to that of an experienced engineer. The nature of this problem depends on the level of the functional description, the set of implementation primitives, and the criteria of acceptability. Early work centered on developing algorithms for translating a boolean function into a minimum two-level network of boolean primitives. Extensions were developed for handling limited circuit fan-in and alternative cost functions [1, 2]. But because these algorithms search for minimal implementations they are necessarily exponential and require too much space and time to be used on most actual designs.

Later efforts attempted to raise the level of specification. The DDL work at Wisconsin [2, 3, 4], APDL at Carnegie-Mellon University [5], and ALERT at IBM all began with behavioral specifications and produced technology-independent implementations at the level of boolean equations. The results were usually more expensive than manual implementations and did not take advantage of the target technology. For example, the ALERT system was validated on an existing design, the IBM 1800, and the implementation produced required 160% more circuits than the manual design [6].

Several tools have been developed at Carnegie-Mellon University to support the early part of the design cycle [7, 8, 9, 10]. In one experiment [11] the CMU-DA system was used to implement the data path portion of a PDP-8/E. They began with a functional description of the machine and produced an implementation in two technologies of the registers, register operators, and their inter-connections, but not the control logic to sequence the register transfers. When the target technology was TTL series modules their design required 30% more modules than the DEC implementation. When it was CMOS standard cells they required 150% more area than an existing Intersil chip.

We are concentrating on the control portion of a machine, since its design is more error prone than data path design. Thus we assume that all memory elements of the final implementation are identified in the specification. Also we are focusing on "random logic" implementations, instead of generating microcode for a control processor or using a programmable logic array. Our implementations are interconnections of primitives selected from a specified set, and connected to satisfy given performance requirements and technology restrictions.

## The Approach

In a previous paper [12] we described what we believe to be a new approach to this form of synthesis. We are not proposing a completely automatic replacement for the manual design process. Instead, we envision an interactive system in which the user operates on a logic design at three levels of abstraction. He begins with a initial implementation generated in a straightforward manner from the specification. He can simplify the implementation at this level, and when satisfied can move to the next level. He does this by applying transformations, either locally or globally, to achieve the simplification or refinement. By being able to operate on the implementation at several levels, the user can often make a small change at one level that will cause a larger simplification at a lower level. By limiting the user to directing function-preserving transformations, we can insure that in all cases the implementation produced will be functionally equivalent to the specified behavior.

Logic synthesis is a problem of finding a feasible (not an optimal) implementation: a network of primitive boxes that satisfies a large number of constraints. In addition to gate and I/O pin limitations, there are timing constraints, a restricted library of primitives, driver requirements, clock distribution rules, fan-in and fan-out constraints, and rules for testability. Since we eventually hope to apply our techniques to VLSI chips, we are attempting to limit our transformations to local changes that do not require time or space exponential in the size of the chip.

CH1511-5/80/0000-0234 $00.75 © 1980 IEEE

## A Prototype System for Logic Synthesis

The organization of the logic synthesis system is shown in Figure 1. Its inputs are the register transfer specification, the interface constraints, and the technology file which characterizes the target technology. The output is a detailed implementation in terms of the primitives of the target technology, which is submitted to placement and wiring programs for physical design. After placement and wiring, some excessively long paths may become apparent. In this case the synthesis process is repeated with a revised specification or modified constraints until an acceptable implementation is achieved.

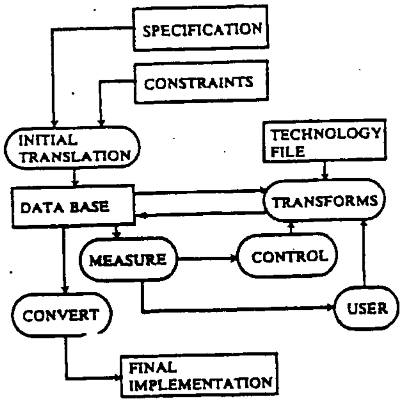


Figure 1. The Logic Synthesis System

## The Logic Synthesis Data Base

An important requirement of our approach is that the data base be capable of representing the implementation at different levels of abstraction. Our system to support logic synthesis makes use of a graph-like internal data structure for storing the implementation as it progresses from the higher-level description to its final form, and all transformations operate on this graph. There is a single organizational component: the 'box'. A box has input and output terminals which are connected by wires to other boxes. Each box also has a type, which may be primitive or may reference a definition in terms of other boxes. Thus a hierarchy of boxes can be used, and an instance of a high-level box such as a parity box can be treated as a single box or expanded into its next level implementation when that is desirable.

The Logic Synthesis data base is implemented using a system originally developed for use in an experimental compiler project within IBM Research [13]. It is made up of two groups of tables. The first group describes the technology being used. For each box type in the technology, the tables contain information such as name, function, and number and names of input and output pins. This data is created in batch mode and is read during initialization of the interactive system.

The second group of tables contains the representation of the logic created by the interactive system. This group consists of a box table, a signal table, and a set of auxiliary tables which describe the relationship between the boxes and the signals. There is some intentional redundancy in the data; each box has a complete list of input and output signals, and each signal has a source and a list of sinks. Every box table entry contains type information which provides a link to the technology group. This allows programs to get technology information about a specific box.

Transformations communicate with the database via a layer of functions which perform all data addition, retrieval, and deletion. They also provide the transformations with the ability to traverse a chip by following signal paths, or by visiting each box. The functions provide a conceptual view of the database which remains stable even when the database is altered. The table structure representing this view can be significantly changed with a minimal impact on the processing programs.

## The Experiments

The first three experiments with the Logic Synthesis System were attempts to produce implementations for chips from existing processors that had been specified functionally and implemented by engineers. The existence of the engineers' implementations permitted comparison of designs and a study of the differences between manual designs and those produced automatically. The fourth experiment was a use of the same system to transform an existing chip implementation into a new technology. Each of the experiments was carried out automatically, although the particular application sequence of transformations was the result of much experimentation.

### Experiment 1

For our first experiment we selected a chip that was characterized as "straightforward". The specification described 7 registers totaling 24 bits, 2 parity operators, and the conditions for the data transfers. The target technology was a TTL masterslice that provided 96 I/O pins and 704 NAND gates on each chip. In addition to the NAND gates, there are a number of macros such as receivers, senders, and flip-flops that are implemented with these NAND gates. Restrictions on the use of the primitives available such as fan-in and fan-out requirements, timing constraints, clocking and powering rules, etc., were described in the technology file or in some cases built into the transformations.

Our objective was to find a set of transformations and a sequence for applying them such that the original functional specification could be transformed by a sequence of small steps into an acceptable implementation. After examining many alternative scenarios, we arrived at the one shown in Figure 2. A similar scenario was used in all the synthesis experiments, with some differences that will be discussed later. We also spent considerable time talking with the designer of this chip to understand the motivation for the many design decisions.

Our strategy is to generate an naive implementation from the functional specification and then to perform local simplifications at three levels of abstraction: the AND/OR level, the NAND level, and the hardware level. The initial implementation is produced by merely replacing specification language constructs with their equivalent AND/OR implementations. Methods for this translation have been described in [3, 5]. Transformations are used at each level to simplify the implementation according to appropriate measures and to move the implementation from one level to the next. The transformations are local in that they replace a small subgraph of the network (usually five or fewer boxes) by another subgraph which is functionally equivalent but simpler according to some measure.

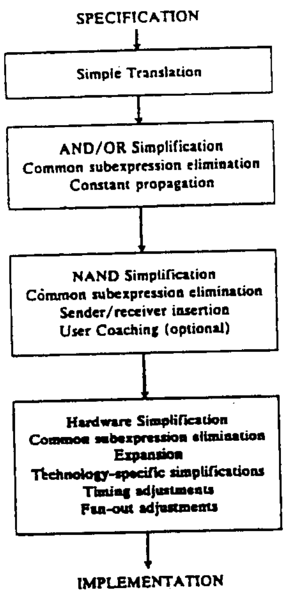


Figure 2. The Scenario for Synthesis

At every level the implementation is a network of boxes connected by signals. At the first level the boxes are of types such as AND, OR, NOT, PARITY, or REGISTER. Simple local transformations are applied to reduce the number of boxes. The particular transformations used are listed below:

NOT(NOT(a)) $\rightarrow$ a
AND(a,NOT(a)) $\rightarrow$ 0
OR(a,NOT(a)) $\rightarrow$ 1
OR(a,AND(NOT(a),b)) $\rightarrow$ OR(a,b)
XOR(PARITY(a1,...,an),b) $\rightarrow$ PARITY(a1,...,an,b)
AND(a,1) $\rightarrow$ a
OR(a,1) $\rightarrow$ 1

The last two simplifications are examples of a more general constant propagation that is performed. This action may leave fragments of logic disconnected. We clean up the dead logic in a manner similar to the way compilers perform dead code elimination. Another technique from optimizing compilers, common subexpression elimination, is also applied here and at other points in the synthesis process to further reduce the size of the implementation.

Next the AND, OR, NOT, and most other operators of the initial description are replaced by their NAND implementations. NANDs were selected for this level because they are the primitives available in this particular technology. This transition also is accomplished by local transformations, and may introduce unnecessary double NANDS, which will be eliminated later. Also at this point, the chip interface information is used to place generic (i.e., not technology-specific) senders and receivers on the chip inputs and primary outputs, and to insert NANDs where necessary to insure the correct signal polarities. Higher-level operators such as EO and PARITY are not replaced at this point but will be expanded at the hardware level.

Simplifying transformations are now applied to each signal in the network at the NAND level. These NAND transformations attempt to reduce the number of boxes of the implementation without increasing the number of connections. To accomplish this, the transformations must check the fan-out of the various signals involved, since this will affect the number of boxes and signals actually removed. The transformations are applied repeatedly throughout the network until no more apply. Figure 3 illustrates the NAND transformations used in this experiment.

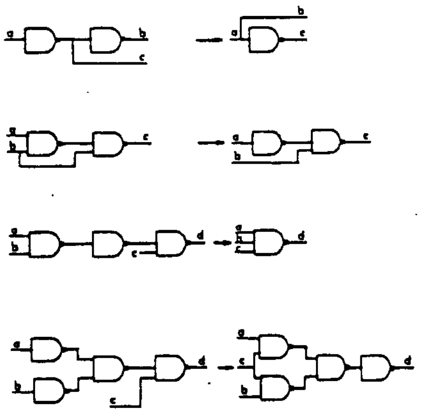


Figure 3. The NAND Transformations

In examining the implementation after the NAND transformations were applied, it was noticed that further improvements could be made. In particular a reduction in fan-out of a signal by repowering its source would allow a transformation to apply and eventually reduce the size of the implementation. The system allows repowering and some other transformations to be applied to particular signals, rather than across the whole implementation, as a form of user "coaching". In this instance coaching saved only four boxes, but that resulted in an implementation slightly better than the manual one.

In the transition to the hardware level the NAND gates and generic registers are replaced by technology-specific primitives. Single primitives or macros are selected to match the fan-in of the actual primitives with that of the "idealized" boxes. Also the number of control and data lines of the idealized registers might exceed those normally available, and necessitate the generation of additional logic. At this point the implementation is in terms of primitives used by the engineers in their implementations, but because transformations have been made locally, there may be some violations of timing, fan-out, and other technology restrictions.

The simplifying transformations at the hardware level are of two sorts. Some are simplifications similar to those at the previous levels, such as eliminating the equivalent of double NOTs, which may occur as a result of expanding higher level boxes, such as EQ and PARITY. Others attempt to take advantage of the particular technology. For example, flip-flops in this technology provide an output and its complement allowing some inverters to be removed at this level. Also, because of combination flip-flop-receiver books available, some receivers may be eliminated. Wired or dotted ANDs can be introduced to reduce cell count where possible. Other technology-specific transformations applied at this level distribute clock signals to flip-flops according to the technology rules, eliminate long and short paths between flip-flops (assuming a unit gate delay and technology-specific guidelines), and adjust fan-out by repowering signals.

Results -- the first experiment resulted in a synthesized implementation that was remarkably similar to the manual one. In fact, it required four fewer cells, five fewer connections, and four shorter paths than the engineer's implementation. The similarity, however, was not such a surprise since we had used this example in the design of our system, and since we had worked so closely with the chip's designer.

## Experiment 2

For our second experiment we wanted to try the same sequence of transformations on a more complex chip. The chip specification we selected contained 13 register bits, a 3 bit counter, a 5 bit counter, 2 parity operators, and more complex conditions controlling the data transfers. The target technology was the same as in the first experiment. Also this time there was virtually no contact with the engineer who designed the chip.

While we tried to use the same scenario, we did make two changes. There was no coaching in this experiment and counters were handled differently from the EQ and PARITY in the first experiment. We found that it is better to expand the counters at the AND/OR level, than at the hardware level. This exposes the expanded counter to all subsequent simplifications and allows one definition to be used for different technologies. The expansion transformation therefore has been extended to permit expansion of a nonprimitive box at any level.

Results -- the synthesis of the second chip resulted in a implementation with 15% more cells and 20% more connections than the manual implementation. We are currently analyzing these results to understand why our implementation is more complex.

## Experiment 3

The third experiment was an attempt to synthesize another complex chip in a different technology. This third chip specification described 28 register bits, 3 parity operators, 4 decoders, 7 comparitors, and even more complex control logic. The target technology was an ECL masterslice. In addition to a new set of technology rules and restrictions, this meant that the basic primitive was a NOR and that each primitive had "dual-rail outputs", that is it provides both polarities of its output. The synthesis scenario was adapted to this technology and changed slightly, but the three levels of implementation were maintained. The decoders and comparitors were expanded at the AND/OR level and the AND/OR transformations remained unchanged. Common subexpression elimination was applied more often at this level and throughout the scenario.

The NAND level became the NOR level because of the new technology. This required a new transformation to translate the AND/OR primitives into NORs, and a set of NOR simplifications. These were originally just the NAND transformations with the NANDS converted to NORs, but we later realized that with dual rail outputs, an apparent box saving at the NOR level might not be a saving at the hardware level, and that the transformation might increase fan-in or number of connections. Thus different fan-out restrictions were used in the NOR transforms. At the hardware level new definitions for PARITY and EQ, were written. The technology-specific transformations had to be rewritten for the new technology, and some new ones were added, such as the one to eliminate inverters.

Results -- this experiment resulted in a implementation with 10% more boxes than the manual one. We are trying to account for this additional logic and determine if it could be eliminated through local transformations.

## Experiment 4

In addition to the 3 synthesis experiments we extended the system to explore transforming a chip implementation from one technology to another (TTL masterslice to masterslice ECL as an example). This required two new transformations, one that transformed primitives at the hardware level back to the NAND level, and a second that transformed the NAND implementation into a NOR one, while preserving the chip input/output behavior. This approach is better than the straightforward replacement of old technology primitives by new ones, since it exposes the remapped implementation to the simplifications at the NOR level and at the hardware level.

Results -- Unfortunately, this chip conversion was not performed manually so we could not make an objective comparison. We did check that the input/output behavior was preserved and showed the implementation to an experienced engineer who found no serious problems.

## Comparing Implementations

One of the problems that confronts us is the difficulty of evaluating the result of the synthesis process. In our work to date, this evaluation has meant a comparison between our generated implementation and a manually produced implementation. There are two aspects to the comparisons that we must perform. One is the problem of determining functional equivalence between the two implementations. The other is to furnish a response to the ill-posed question: "How do these implementations differ?"

Functional equivalence in its full generality is the problem of boolean equivalence and known to be co-NP complete. This implies that at our present level of understanding, it is not possible to devise a program which will efficiently, in all cases, decide equivalence between two implementations. We cannot solve this problem; but we are exploring heuristics which may offer significant speed-up on a large class of implementations. A report on this work is in preparation.

Even when two implementations are functionally equivalent, we are still interested in their structural similarity. This form of comparison permits us to evaluate stylistic difference between our implementation and that produced by an engineer. This is necessary for discovering new heuristics. For this form of comparison we are considering formalizing the notion of "distance" between two implementations, following an analogy to the spelling correction problem.

## The Future

During the coming months we plan to continue analyzing the results of our experiments to determine what improvements should be made to our system. We will also look at more ambitious chips -- chips that have required minimization or that have caused long path problems, when implemented manually. We hope to arrive at a set of measures and transformations that will provide acceptable implementations for a large class of examples. In addition, we will explore:

- multi-chip synthesis -- starting with a functional specification that requires several chips, developing additional measures and transformations that will trade resources across chip boundaries.
- engineering changes -- examining how such a synthesis system could respond to EC's where minimum, local changes are highly desirable.
- transformation specification -- looking at how transformations could be described at a high level and compiled for efficient application.
- transformation correctness -- considering what properties of transformations such as function-preserving should be proved and demonstrating how such proofs can be accomplished.

## Summary

We are in the process of exploring what we believe is a new approach to the old problem of logic synthesis and are encouraged by our initial experiments. We have built a prototype synthesis system and used it to synthesize 3 masterslice chips, requiring 0%, 15%, and 10% more logic than their respective manual designs. We have also used our system to remap an implemented chip into a new technology, while preserving its input/output behavior. We plan to perform further experiments, to study the remaining differences between the automatic and manual implementations, and to improve the competence of our prototype system. Our hope is that computationally manageable techniques based on local transformations can be applied to improve naive implementations to acceptable ones. This could greatly shorten processor development and validation times.

## Acknowledgement

We would like to thank William van Loo and James Zeigler for many helpful discussions on masterslice chip design. Also John Gerbi, Thomas Wanuga, and Alan Stern have made valuable contributions to the design and implementation of the prototype synthesis system.

## References

[1] Breuer M A (Editor), *Design Automation of Digital Systems*, Prentice-Hall, 1972.

[2] Dietmeyer D L, *Logic Design of Digital Systems*, Allyn and Bacon, 1971,1978.

[3] Duley J R, "DDL -- A Digital Design Language", Ph.D. Thesis, University of Wisconsin 1968.

[4] Duley J R and D L Dietmeyer, "Translation of a DDL Digital System Specification to Boolean Equations", *IEEE TC* Vol C-18, April 1969.

[5] Darringer J A, "The Description, Simulation, and Automatic Implementation of Digital Computer Processors", Ph.D. Thesis Carnegie-Mellon University 1969.

[6] Friedman T D and S C Yang, "Quality of Design From an Automatic Logic Logic Generator(ALERT)," Proc. 1970 Design Automation Conference.

[7] Barbacci M, "Automated Exploration of the Design Space for Register Transfer Systems", PhD Thesis, CS Dept., Carnegie-Mellon University, 1973.

[8] Thomas D E, "The Design and Analysis of an Automated Design Style Selector", PhD Thesis, EE Dept., Carnegie-Mellon University, 1977.

[9] Snow E A, "Automation of Module Set Independent Register-Transfer Level Design", PhD Thesis, EE Dept., Carnegie-Mellon University, 1978.

[10] Hafer L J and A C Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process", Proc. 15th Design Automation Conf., 1978.

[11] Parker, A., et.al., "The CMU Design Automation System -- An Example of Automated Data Path Design", Proc. 16th Design Automation Conf., 1979.

[12] Darringer J A, and W H Joyner "A New Approach to Logic Synthesis", 17th Design Automation Conf. June 1980.

[13] Allen F, et.al "The Experimental Compiler System", IBM Journal of Research and Development, Nov. 1980.

14

# C. .D Systems for IC De.ign

MARVIN E. DANIEL AND CHARLES W. GWYN, SENIOR MEMBER, IEEE

*Abstract*—As integrated circuit (IC) complexities increase, many existing computer-aided design (CAD) methods must be replaced with an integrated design system to support very large scale integrated (VLSI) circuit and system design. The framework for a hierarchical CAD system is described. The system supports both functional and physical design from initial specification and system synthesis to simulation, mask layout, verification, and documentation. The system is being implemented in phases on a DECSystem 20 computer network and will support evolutionary changes as new technologies are developed and design strategies defined.

## INTRODUCTION

COMPUTER-AIDED DESIGN (CAD) of integrated circuits (IC's) has had various interpretations as a function of time and definition source. These interpretations range from use of simple, interactive graphics and digitizing systems to individual programs used for circuit or logic simulation, mask layout, and data manipulation or reformatting. In many instances, the word "aided" is deemphasized to imply nearly automatic design. Within the context used in this paper, CAD refers to a collection of software tools to provide the designer with design assistance during each phase of the design. Although many decisions are made by the software during the design process, important decisions are the designer's responsibility. The computer aids or tools provide the designer with a rapid and orderly method for consolidating and evaluating design ideas and relieve the designer of numerous routine and mechanistic design steps.

### *Background*

A brief review of some of the techniques and terminology that evolved into today's CAD is instructive to viewing the collection of tools available today. CAD had its origins in the mid-to-late 1950's. With the advent of high-speed digital computers, some of the pioneering work of Kron [1], [2] was applied to the simultaneous solution of network equations. This formulation was used, in part, by computer codes such as NET-1 [3] (Network Analysis Program) and ECAP [4] (Electronic Circuit Analysis Program). From a physical point of view, solution of the network problem predicts the behavior of a system in terms of the element characteristics and interconnections. Viewed as a mathematical problem, the properties of a topological structure (linear graph) and a superimposed algebraic structure (interrelations of the nodes, branches, and meshes of the graph) are determined.

This early work of solving steady-state and transient netw problems was the genesis of circuit simulation and was de nated CAD. By the early 1970's, the growing need to simul large circuits and thus obtain the detailed solution of a la number of partial differential equations forced the develo ment of optimization methods and higher levels of abstracti to represent physical systems. Whereas very detailed mod are used to simulate the behavior of individual transisto more abstract representations are used for timing and lo simulation. A timing simulator uses only current-volt tables for transistor models; capacitive loading, and circ connectivity to determine the signal waveforms at each c cuit node. Logic or gate-level simulation solves the equi lent Boolean equations with delay elements inserted betw gates to account for signal timing.

The use of computer aids in the layout of IC masks ess tially proceeded along two approaches: interactive graphic sy tems and automatic layout based on standard cells. Early i teractive graphic systems provided a method for capturing d design by recording coordinate information by manually di tizing and editing data. Methods for superimposing m layers, scaling, enlarging, contrasting, and reviewing the res were expanded to provide fast, sophisticated drawing co mands for constructing, editing, and reproducing complex f ures; performing dimensional tolerance checks; selective era expand, move, and merge; symbolic input; pattern generatio etc.

Automatic layout methods have classically relied on a libra of circuit components or cells in the form of mask geometr defining logic gates. Most software required cells with st dard heights and varying width. The layout software, pl cells in rows, attempts to optimize the cell position in the ro and interconnects the cells in wiring channels between the c rows. The automatic standard cell layout programs h evolved from simple linear cell placement in a single ro which was subsequently folded to fit a square area [5], complex placement in two dimensions. The early standard c layout aids supported the use of single entry (connections one side of the cell) cells placed in the row in a back-to-ba configuration. Power was distributed to the cells through t common connection on the backside of the cell. Newer st dard cell layout programs support cells containing termi on both sides of the cell. Instead of placing cells in a back-t back configuration, each cell row contains a linear placeme of the cells [6], [7] and is separated by wiring channels.

The early use of CAD for physical design verification co sisted of performing simple design rule checks for width a spacing violations on mask artwork files. Functional integri was verified through circuit simulations.

Manuscript received April 10, 1981; revised September 8, 1981. This work was supported by the U.S. Department of Energy.

The authors are with Sandia National Laboratories, Integrated Circuit Design, Dept. 2110, Albuquerque, NM 87185.

### *Initial Sandia Design Aids*

After the initial decision to establish a CAD capability for integrated circuits at Sandia, software was obtained from universities and private industry wherever possible to provide a basic capability. For example, the SPICE [8] circuit analysis code was obtained from the University of California at Berkeley, and the PR2D [6] standard cell layout program for metal gate cells was obtained from RCA. Since Sandia had previously developed a simple logic analysis capability, this work was accelerated to provide gate-level simulation. Software was developed to postprocess the mask layout data to pattern generator formats for mask generation and plot file information for the Xynetics flatbed plotter. A software package was purchased from Systems, Science, and Software, Inc., and installed on an existing interactive graphic system. In addition, translator subroutines were written to convert design information data formats to minimize the manual data translation and to allow the design information to be added only once in the system. The above process is oversimplified, since a substantial commitment of staff was required to: (1) understand the acquired software and debug and correct errors, (2) perform modifications to support the software on Control Data computers, (3) identify and develop required translator software, and (4) develop additional design aids. These problems and the subsequent software modifications required a large manpower investment, since most of the software was not documented, had evolved over many years with several authors, used nonstandard Fortran, and was unstructured.

### *CAD System Requirements*

Although the initial design aids provided a valuable capability for simple metal gate CMOS custom IC design, many deficiencies were identified. These deficiencies coupled with the need to support new technologies with shrinking design rules (and thus rapidly increasing circuit complexities), new design styles, and changing design objectives, required a new approach to the development of aids for IC design.

Several general objectives for new aids were identified. The new aids must (1) be user oriented, (2) use modular software, (3) be evolutionary to meet changing design needs, and (4) be integrated into a complete design system rather than exist as an independent collection of disjoint tools.

Computer aids must support both functional and physical design. Functional design aids include synthesis, verification, simulation, and testing at architecture, system, logic, circuit, device, and process levels. Physical design aids support partitioning, layout, and topological analysis at all design levels. Functionality, testability, and physical design must be considered in parallel throughout the design process.

All CAD software must be as technology independent as possible and support various levels of designer sophistication from inexperienced first-time users to state-of-the-art system designers. To meet these goals, the interaction between individual programs, data base, and design engineer must be through a single, consistent interface. This interface should support monitoring the progress of a design, supply options at any stage in the design process, and assist in design documentation and maintenance.

TABLE I

| Complexity | Examples |
|---|---|
| VLSI ($\sim 10^5$ devices) | Microcomputers, cryptographic circuits, ROMs, RAMs |
| LSI ($\sim 10^4$ devices) | Microprocessors, A/D & D/A converters, ALUs, FFT circuits, ROMs, RAMs |
| MSI ($\sim 10^3$ devices) | Adders, complex gates, multiplexers, ROMs, RAMs |
| SSI ($\sim 100$ devices) | AND, OR, NAND, NOR gates, buffers, memory cells |
| Primitive elements | Transistors, resistors, capacitors |

A system design language—or Hardware Description Language (HDL)—must be available for describing all levels of system behavior and structure. Organized in a hierarchical manner, the language should support functional and physical descriptions and the relationships among entities.

Synthesis aids must facilitate the addition of sufficient detail to generate a complete system description. Design verification software monitors internal consistency and completeness of system specifications. Final system specifications should be retained in a dynamic data base providing files in the proper format for input to each of the design aids and for documentation.

A complete CAD system satisfying the above requirements has been designed and implementation is in progress. The basic system description has been divided into three sections. The first section describes design flow and the concept of top-down design with bottom-up implementation. A hierarchical design approach is used with appropriate merging of levels to accomplish the design of VLSI systems. Requirements for specific computer aids are outlined in the second section. The final section describes the implementation philosophy, computer hardware, and present software development status.

## Hierarchical Design

The CAD system supports a number of functions at each level in the design hierarchy. Design proceeds in a top-down sequence with bottom-up detailed implementation and addresses both functional and physical problems at each level.

Architecture, the top level of the design hierarchy, contains elements for a broad functional system description, requirements for interfacing units specifying performance and compatibility, and methods for partitioning the system into major functional blocks such as processors, memory, and I/O. The architectural specification can be expanded at the system level to generate a more detailed description consisting of register transfer level subsystem functions.

At the logic level, system functions are defined as interconnections of fundamental gates or modules. Logic modules can be further decomposed into circuit primitives (e.g., transistors, resistors, capacitors). Finally, in order to construct circuit elements and determine their behavior, the physical implementation and process technology may be considered.

To support a variety of technologies in a hierarchical design structure, a data base consisting of a library of elements of varying sophistication must be maintained. A distinct library must exist for each technology used. Typically, the hierarchi-

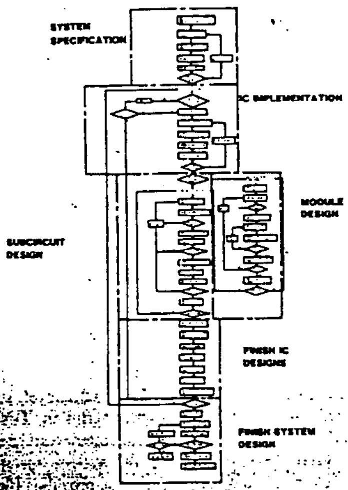
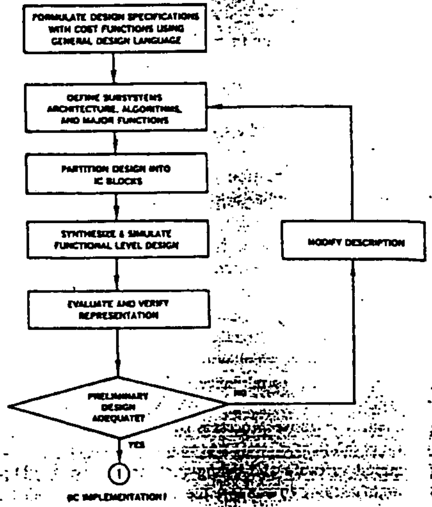


Fig. 1. System design sequence.



FORMULATE DESIGN SPECIFICATIONS WITH COST FUNCTIONS USING GENERAL DESIGN LANGUAGE
DEFINE SUBSYSTEMS ARCHITECTURE, ALGORITHMS, AND MAJOR FUNCTIONS
PARTITION DESIGN INTO IC BLOCKS
SYNTHESIZE & SIMULATE FUNCTIONAL LEVEL DESIGN
EVALUATE AND VERIFY REPRESENTATION
MODIFY DESCRIPTION
PRELIMINARY DESIGN ADEQUATE?
NO
YES
1
(IC IMPLEMENTATION)

Fig. 2. System specification and partitioning.

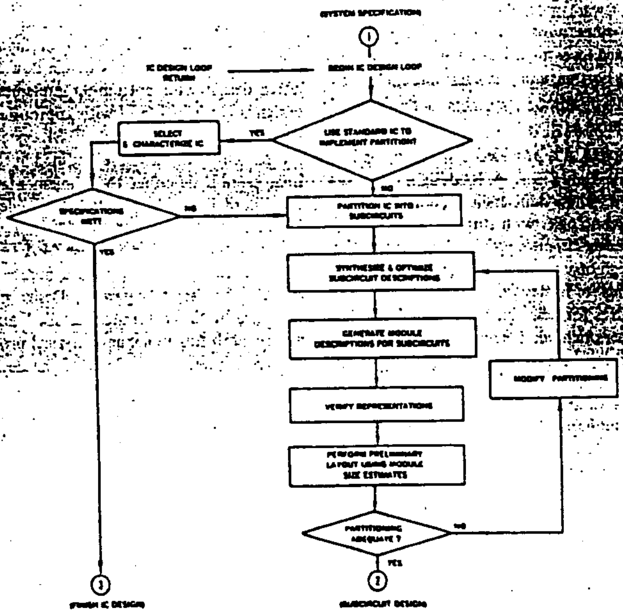


Fig. 3. IC design planning.

cal library will contain circuits and subcircuits at various levels of complexity (VLSI, LSI, MSI, SSI, circuit primitives and/or discrete devices) as defined in Table I.

Electronic system design using computer aids can be divided into six major design sections: (1) system specification and partitioning, (2) IC design planning and initial implementation, (3) subcircuit design, (4) module design, (5) completion of individual IC design, and (6) completion of system design. A design flow diagram summarizing these major functions is shown in Fig. 1.

DEF011953

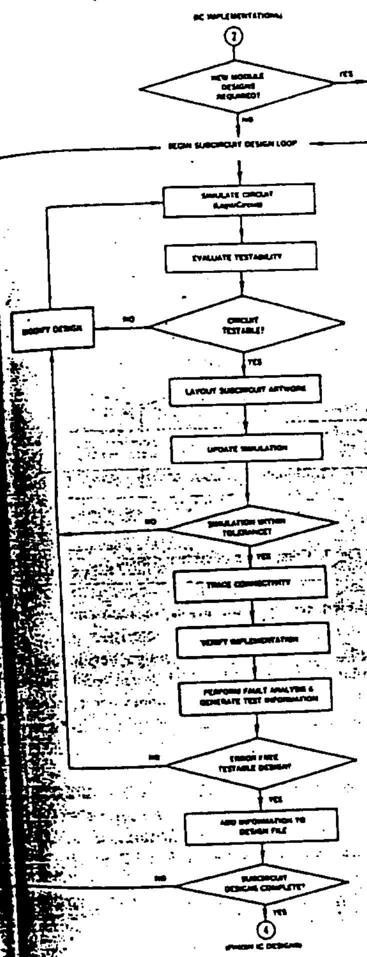

IC IMPLEMENTATION
2
NEW MODULE DESIGNS REQUIRED?
YES
DESIGN MODULES
NO
BEGIN SUBCIRCUIT DESIGN LOOP
SIMULATE CIRCUIT (Logic/Circuit)
EVALUATE TESTABILITY
CIRCUIT TESTABLE?
NO
MODIFY DESIGN
YES
LAYOUT SUBCIRCUIT ARTWORK
UPDATE SIMULATION
SIMULATION WITHIN TOLERANCE?
NO
YES
TRACE CONNECTIVITY
VERIFY IMPLEMENTATION
PERFORM FAULT ANALYSIS & GENERATE TEST INFORMATION
ERROR FREE TESTABLE DESIGN?
NO
YES
ADD INFORMATION TO DESIGN FILE
SUBCIRCUIT DESIGNS COMPLETE?
NO
YES
4
(FINISH IC DESIGN)

Fig. 4. Subcircuit design.

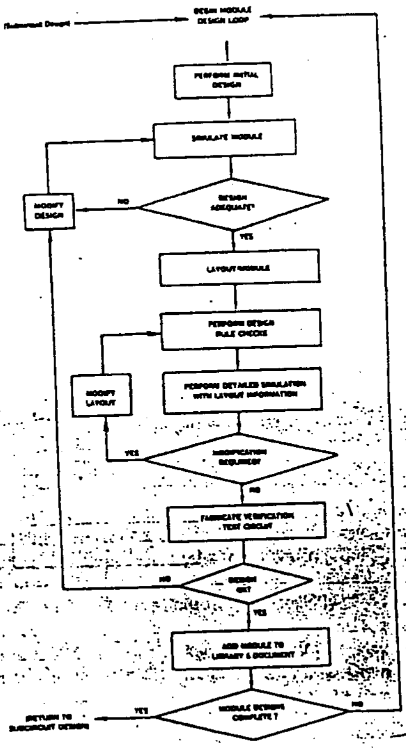


Fig. 5. Module design.

The flow diagram in Fig. 1 is subdivided and enlarged to ow the individual design steps in Figs. 2-7. The first major p consists of developing system specifications and partition- guidelines (Fig. 2). After developing general specifications, systems, architecture, algorithms, and major functions are fined. Each subsystem is partitioned into IC blocks which synthesized and simulated at the system design level. This liminary design is evaluated and modified if necessary by peating the above steps.

The next major step in the design is implementation of each based on the system specification (Fig. 3). For a general ctronic system, the first decision to be made is whether tom or commercial circuits will be used. If a characterized mercial circuit is available in the library, the circuit is selected. If the required commercial circuit is not in the library, the specifications must be obtained and compared with the system specification. If the system specifications are met, the commercial circuit can be used; if not, a custom circuit must be designed.

The first step in designing a custom IC is partitioning the IC into subcircuits. Each subcircuit is synthesized and optimized. Module descriptions for each subcircuit are developed and verified, and a preliminary layout using the estimated module sizes is performed. At this point, a decision can be made concerning the adequacy of the partitioning. If the preliminary layout does not conform to the size restrictions for the circuit or if during the circuit verification, signal-delay paths are longer than desired, the partitioning must be modified and the circuit design repeated.

If partitioning is acceptable, the subcircuit design step can be initiated (Fig. 4). For each subcircuit design, a decision must be made concerning the adequacy of modules in the library for implementing the design; new modules must be designed and chracterized as required (Fig. 5).

During subcircuit implementation, each circuit can be simulated at various individual or combined simulation levels, the testability evaluated, and the artwork generated. After art-

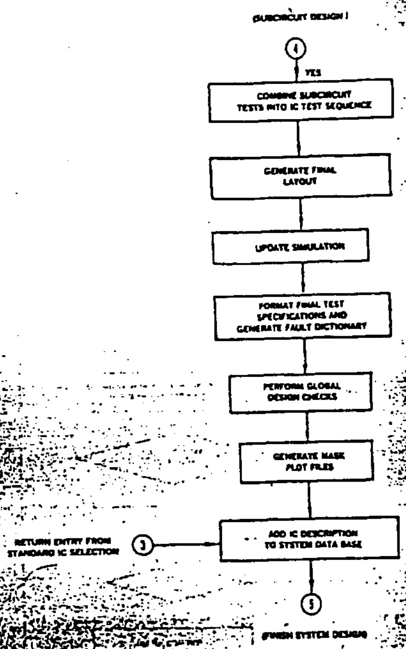

(SUBCIRCUIT DESIGN)
4
YES
COMBINE SUBCIRCUIT TESTS INTO IC TEST SEQUENCE
GENERATE FINAL LAYOUT
UPDATE SIMULATION
FORMAT FINAL TEST SPECIFICATIONS AND GENERATE FAULT DICTIONARY
PERFORM GLOBAL DESIGN CHECKS
GENERATE MASK PLOT FILES
ADD IC DESCRIPTION TO SYSTEM DATA BASE
RETURN ENTRY FROM STANDARD IC SELECTION
3
5
(FINISH SYSTEM DESIGN)

Fig. 6. Completion of IC design.

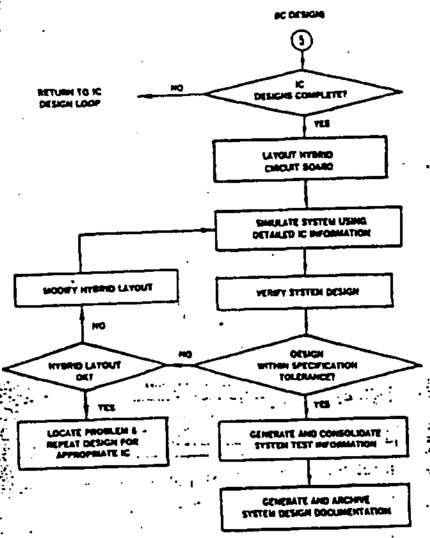

IC DESIGN
5
IC DESIGNS COMPLETE?
NO
RETURN TO IC DESIGN LOOP
YES
LAYOUT HYBRID CIRCUIT BOARD
SIMULATE SYSTEM USING DETAILED IC INFORMATION
MODIFY HYBRID LAYOUT
VERIFY SYSTEM DESIGN
NO
HYBRID LAYOUT OK?
NO
DESIGN WITHIN SPECIFICATION TOLERANCE?
YES
YES
LOCATE PROBLEM & REPEAT DESIGN FOR APPROPRIATE IC
GENERATE AND CONSOLIDATE SYSTEM TEST INFORMATION
GENERATE AND ARCHIVE SYSTEM DESIGN DOCUMENTATION

Fig. 7. Completion of system design.

work design, the simulation net list must be updated to include circuit parasitics associated with the specific layout. This simulation also provides information concerning circuit output drive and power required for subcircuit operation. If the simulation is within tolerance, static design verification is initiated; connectivity is traced; and the layout is checked for conformance to design rules. The layout and associated simulation characteristics are compared with the initial circuit specifications. Finally, test information is generated for the subcircuit.

After each subcircuit design has been completed and verified, the circuit design information is added to a design file in the data base. The next major step consists of completing the IC design by combining subcircuit design information (Fig. 6). The IC layout is assembled by placing and interconnecting subcircuit layouts. The simulation is updated to assure correct circuit operation, test specifications are generated, and a fault dictionary is developed for system diagnosis. Physical design verification, including global rule checks and connectivity checks, is performed, and the mask information for each IC is generated. Finally, design information for each IC is added to the system data base.

The last major step in the design sequence consists of completing the system design (Fig. 7). After selecting commercial circuits or designing custom IC's, layouts are performed for the system circuit boards. After the printed- or hybrid-circuit board layouts have been completed, the entire system is simulated, using detailed design information for each circuit. Th simulation determines the electrical characteristics of the sy tem and provides a final functional design verification b comparing simulation results with initial design specification Test sequences for the individual IC's are combined to produ system tests at the printed-circuit board or hybrid-circuit leve During the final step, system design documentation is generat and archived. Generating system documentation consists o obtaining information from the design data base and conso idating it into the appropriate format. In addition to the I mask plot files and test-sequence information, the docume tation includes system performance information with tole ance specifications for acceptance of the final system.

The complete CAD system for IC design will include a num ber of computer programs. These aids can be categorized follows: design specification and partitioning, system and ci cuit synthesis, system partitioning, simulation at various leve IC mask layout, design verification, testability evaluation, te sequence generation, data base, and design documentation.

## Specific Computer Aids Requirements

In addition to a number of specific computer programs fo supporting a hierarchical design, an extensive data base an support software for monitoring the design flow and comm nication are required. Requirements for specific programs an interfaces are outlined below.

### *Design Executive*

Currently, most design automation programs exist as ind pendent entities with idiosyncratic user interfaces and inco patible I/O structures. A Design Executive can provide a sing consistent interface between a user and the complete set of computer aids available on the CAD system.

The Design Executive supports the complex CAD system structure by mediating interactions between the user and the system, the system and the data base, and the user and the data base. The executive system user interface supports comprehensive "HELP" capabilities, the HDL, and a logically integrated command language. Documentation describing the system and the CAD codes is maintained by the Design Executive in a hierarchical configuration. User access to this documentation structure is facilitated by the intelligent intervention of the Design Executive.

### HDL–Design Specification

The structure and behavior of a digital system must be described in various ways at many levels to completely characterize a design. Existing "languages" such as ISP [9], DDL [10], SDL [11] and are usually associated with specific descriptions of architecture, system behavior, system structure, logical structure, circuit structure, logical behavior, or physical structure. These descriptions lack the generality required to support VLSI design, since most are applicable to one level of design and a particular design style and are not integrated into a complete design system.

Ultimately, a comprehensive HDL must be used to describe all levels of system behavior and structure; including normal and faulty electrical operation, logical and functional behavior, and physical structure. Organized in a hierarchical manner, the HDL should allow description of functional and physical entities and relationships among entities.

### Synthesis and Functional Verification

Functional synthesis begins at the architecture level with a description of the overall behavior of a large system and interaction with the environment. As synthesis proceeds, more structural detail is added at system, logic, and circuit levels in the form of interconnection structures. Concurrently, behavior specifications are developed at each level of the design hierarchy.

During synthesis, the actual behavior of the system must be forced to match the specifications. To effect the match, models are required for computational algorithms and interpreters, and procedures for mapping one into the other. Models must be expressible in a computer-readable form in order to manage complexity with an interactive computer support system, permit the separation of structure from associated behavior, and support direct fabrication of the synthesized system.

### Simulation

Simulation, or dynamic design verification, is the process of calculating the behavior of a system within an environment specified by the designer. The objective of simulation is to verify that the system will perform correctly in an operational environment.

Ideally, simulation should be multilevel; i.e., consider larger circuits at a less detailed level with the capability of simultaneously simulating subcircuits more exactly. This concept complements the hierarchical design implementation. In a hierarchical simulator, the basic elements are more complex than simple modules or gates. Models are formulated for entire IC's and include gate, function, and transistor models. Parts of a system are simulated in great detail while other parts are simulated abstractly. Models are written in high-level languages, with models for MSI or LSI blocks only slightly more complex than the model for a simple NAND gate. Thus a system of many thousands of devices or gates can be reduced to a few hundred hierarchical models, making it feasible to simulate entire VLSI systems.

### Testability

Testing is the process of verifying that a system is operating properly. Because of the increased complexity of VLSI circuits, the difficulty of testing is substantially increased. To ensure that a circuit can be tested, design-for-testability techniques must be used throughout the design process.

Approaches to design-for-testability fall into two major categories: testable design styles and testability measure analysis. The use of a testable design style guarantees that generating tests for a circuit will not be impossible. Testability measure analysis computes the difficulty of controlling and observing each internal node from primary input or output pads. This information is used in the design process to locate potential circuit testing problems and provide feedback about the effect of circuit modifications on testability.

In developing a test sequence, knowing that a good test exists is not the same as knowing the test. Deriving a test is the task of the test sequence generation phase. Given a digital circuit and a set of possible faults, a series of input signals (vectors) is generated that will force any faulty circuit to behave differently from the fault-free circuit. The test sequences depend not only upon the intended function of the circuit but also upon the fault set assumed. Test-sequence generation must proceed concurrently with the hierarchical design process.

### Physical Design Aids

Physical design aids include tools to partition, place, and interconnect circuit components and verification tools to ensure the synthesis has been performed properly. To provide an optimum system, physical design must be considered in parallel with functional design and testability.

*Partitioning*–Partitioning programs operate on both functional and physical entities. Partitioning techniques must function in both the initial design and detailed implementation phases. During top-down design, partitioning aids are needed for hierarchical decomposition. During bottom-up implementation, partitions are modified based on lower level implementations.

In addition to optimizing a circuit element assignment based on size and external pin connections, parameters such as circuit speed, power dissipation, and functional groupings must be considered. The partitioning aids must be capable of optimizing any of these quantities as a function of the others.

*Symbolic layout*–Provides a shorthand method for manually sketching a circuit layout using specified symbols to represent various types of transistors and interconnections. The initial layout can be performed on grided paper or directly on an interactive graphics system. During layout, the designer is not concerned with the geometric design rules. Symbolic layouts are postprocessed using computer aids which expand and relocate all symbols and interconnects based on the geometric design rules for the specified technology to provide mask artwork files.

*Physical layout*—The physical layout phase of IC design involves positioning and interconnecting electrical components. In hierarchical design, the concept of a component is generalized. Components may range in complexity from primitive transistors and fundamental gates to microprocessors that can be combined to form a microcomputer.

Hierarchical layout is applicable to a wide range of physical design levels. At the highest design level, the shape of LSI components may be adjusted and combined to form a VLSI circuit. At the lower design levels, the same algorithms may be used to combine gates into registers or transistors into gates.

*Topological analysis*—Because of the high costs and long lead times involved in the fabrication of IC's, it is important to verify the correctness of a circuit design before manufacture. Verification tools must ensure correct physical mask layout, functional operation, and that electrical characteristics are within specified tolerances.

Topological analysis tools can be used to verify correctness of physical layout data by examining the artwork data and the circuit inputs and outputs. Design-rule checking codes perform geometrical, logical, and topological operations on artwork data and compare the results with design rules for the specified technology. Connectivity and electrical parameter data are used to reconstruct a detailed circuit including parasitic electrical components. This reconstructed circuit can be used in performing an accurate functional and timing analysis.

### *Circuit Design Documentation*

After a circuit or system design has been completed, the design can be documented by collecting information from the data base generated during the design process. Design documentation includes information for fabricating the IC's and system as well as information for archiving the completed design to support later modifications. Computer aids should collect manufacturing and archival information and generate appropriate files, specifications, and reports.

### *Data Base*

In general, the distinction between programs and data is that programs are active, data is passive (i.e., programs operate on data). The necessity for supporting the initial design, design changes, and providing software transportability dictates consistency in data handling procedures and mechanisms. Therefore, an efficient data base is vital for coordinating the various functions. As systems become more complex, it is imperative that an overall data base be established and maintained to ensure consistency in design and to eliminate duplication of information.

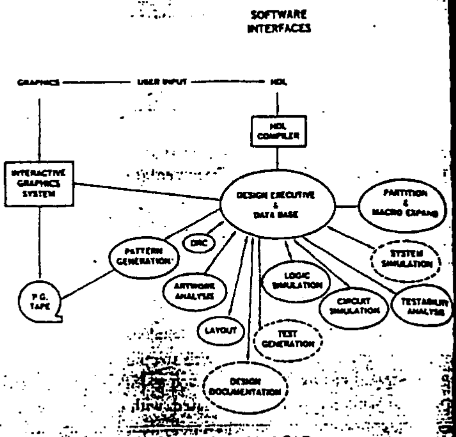


Fig. 8. Sandia hierarchical CAD system.

The data base must be designed in a flexible and modul
fashion to provide upward compatibility with computer har
ware and the evolving CAD software. Modularity is importa
where various portions of the data base may reside at mo
than one computer node. In addition to the actual data us
by CAD programs, the data base should contain all necessa
information to properly document elements being designed.

## Sandia CAD System Implementation

A CAD System meeting the criteria outlined above is be
implemented in phases at Sandia National Laboratories to pr
vide design capability for LSI circuits initially, with evo
tionary expansion and enhancement as required to suppo
VLSI designs.

The design procedures consist of: (1) identifying critical ai
in the design process based on user needs, (2) acquiring exi
ing software when available and subsequently modifying it
meet system and user needs, (3) developing new aids w
appropriate expansion capabilities to support hierarchical d
sign, and (4) integration of all aids into the design syste
framework. A block diagram outlining the present system i
plementation is shown in Fig. 8. The solid blocks represe
areas where design support is presently available, and th
dashed blocks indicate work in progress. Brief descriptions
the hardware and software are given below.

### *Hardware*

The software is supported on a dedicated DECSystem 2
computer system containing 1 million words of main memor
160 million words of disk files, and two 9-track 800/160
bit/in, 75 in/s tape drives. Two Applicon 860 interact
graphics systems and a Versatec 36" electrostatic plotter a
connected to the DECSystem 20. A VAX 11/780 compu
containing 4 megabytes of main memory, 323 megabytes
disk files, and 2 9-track 800/1600 bit/in, 125 in/s tape dri
is interfaced in a network configuration to provide additio
computing capability. Communication among the comput

and the Applicon systems is supported by DECnet. Both dial-up and direct-wired access to the system provide support for alphanumeric and graphics terminals.

*Design Software*

A Universal HDL (UHDL) is being developed in essentially three stages consisting of: (1) language formation by phrase concatenation, (2) redundancy removal, and (3) consolidation and rewrite of the compound language. During the first step, the SDL [11] language (a PASCAL-like description) is used to describe system structure. Other languages are appended in phrase form to describe functional behavior. After gaining experience with the composite languages and after the deficiencies and required enhancements have been identified, a language for the initial UHDL will be written. Continuous refinement and enhancement of the UHDL will be required to meet design and documentation needs.

Translation from UHDL descriptions (initially SDL) to the required formats for simulation, layout, and verification programs is performed by the Design Executive. The Design Executive also manages the data flow between the Data Base files (outlined below) and each of the design aids.

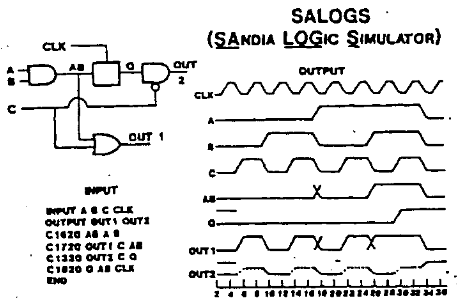
Design synthesis aids currently available include MINI [12] and DDA [13] (Digital Design Aids). MINI, a Programmable Logic Array (PLA) minimization program uses a branch-and-bound algorithm to produce the optimal two-level realization of a set of Boolean equations. DDA provides synthesis of a sequential-state machine based on flow diagram, state assignment, and flip-flop type.

Logic-level design consists of defining, in complete detail, all components (gates, cells, modules, etc.) and interconnections required to implement a specified function. The SALOGS [14] (SAndia LOGic Simulator) program performs true-value 4-state logic simulation (true, false, undefined, and high impedance) and 8-state timing simulation (four states plus transition to each of the four states) as well as states-applied and gate activity analysis and fault simulation. SALOGS is used for dynamic design verification, test sequence evaluation, and fault coverage calculations. Race and hazard analysis is available and the program is capable of using library or user-defined models.

A typical input/output for SALOGS is shown in Fig. 9 for a small logic circuit. The input consists of a connectivity description of logic gates or cells contained in the model library. The plotted output describes the logic signal changes as a function of time for each of the nodes indicated.

Although approximate timing simulation is accomplished in SALOGS through the use of extra states to represent logic values in transition, more exact timing simulation is possible with MOTIS-C [15] and SIMPIL [16] for MOS and $I^2L$ circuits, respectively. These programs use tables to describe active devices in a circuit instead of models with complex equations. Both programs provide more accurate analysis than logic simulators and run more efficiently than circuit simulation codes such as SPICE.

A more exact circuit analysis is obtained by using the SANCA (SANdia Circuit Analysis) program circuit simulation program based on SPICE. This program simulates MOS and bipolar



```
INPUT A B C CLK
OUTPUT OUT1 OUT2
C1620 AB A B
C1720 OUT1 C AB
C1320 OUT2 C Q
C1820 Q AB CLK
END
```

Fig. 9. Typical logic simulation input and output using SALOGS.

circuits, including analog and digital circuits, by numerically solving the network equations to calculate the desired voltages and currents. The cost of increased accuracy (over timing simulation) is a decrease in both execution speed and circuit size that can be analyzed. Model libraries [17] for circuit analysis are maintained to reflect varying levels of accuracy and different modeling philosophies. Circuit models based on device physics, empirical behavior, and hybrid approaches are all used in the circuit simulation environment. SANCA is an enhanced, quasi-interactive, topological analysis program containing model information for specific Sandia technologies and output plot routines.

Physical mask layout is performed using the SICLOPS [18], [19] (Sandia IC Layout Optimization System) and SLOOP [20] (Standard cell LayOut Optimization Program) programs to automatically place and interconnect generalized circuit elements. Two different forms of mask layout are used. A hierarchical mask layout system has been developed and used for specific designs. Hierarchical layout is applicable to a wide range of design levels. At one extreme, LSI components can be combined to form a VLSI circuit. At the other extreme, the same algorithms can be used to combine gates into registers based on a standard cell layout. This hierarchical approach relies on the use of SICLOPS to perform an automatic layout of an integrated circuit using rectangular-shaped blocks. The blocks must be rectangular and can be of arbitrary size and aspect ratio. The program automatically places the blocks and performs the interconnections. Each of the blocks can contain nested subblocks to any depth to provide a hierarchical layout. SICLOPS can also be used to layout a thick film hybrid circuit, as shown in Fig. 10.

The more conventional layout using standard cells placed in rows with interconnection between cell rows in routing channels is supported by SLOOP. In addition to designing modules used by SICLOPS, conventional standard cell IC layouts can be designed. The basic program can be used in an interactive mode to optimize the design, will perform a 100-percent completion of all interconnections by expanding the chip size as required to complete interconnections, contains hierarchical interfaces for use with the SICLOPS code, and supports multi-port gates or cells. A typical IC design using SLOOP is shown in Fig. 11.

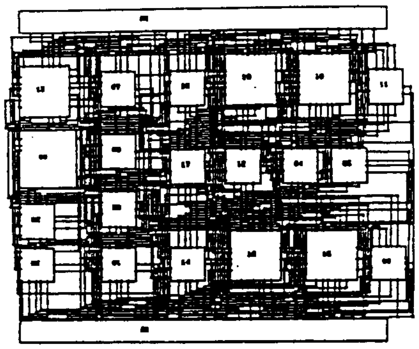


Fig. 10. Thick-film hybrid circuit layout obtained using SICLOPS.

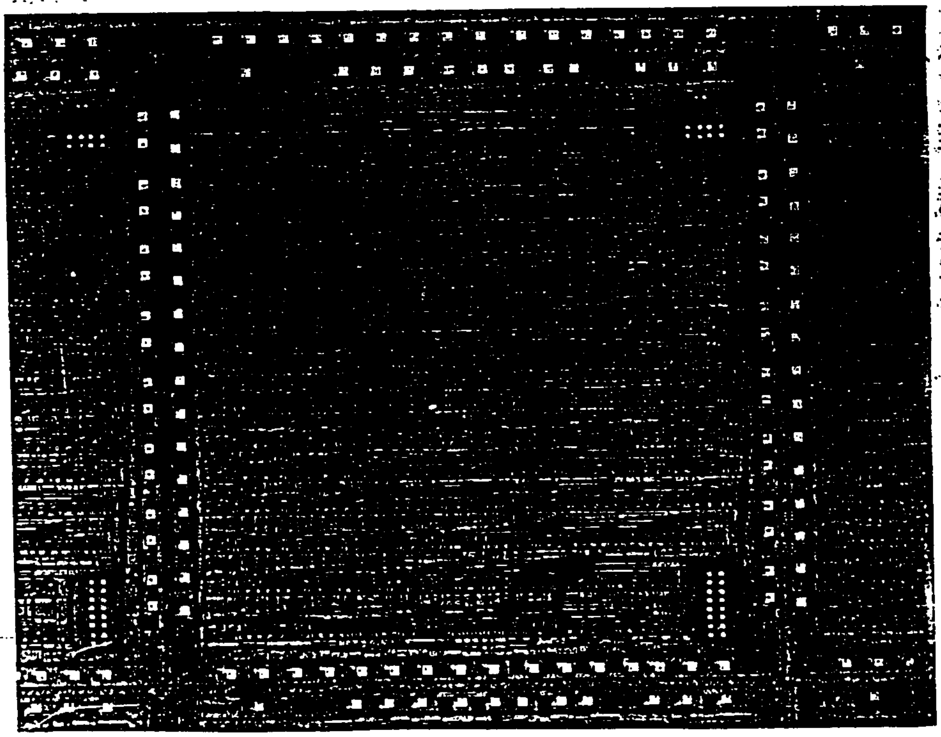


Fig. 11. Typical standard cell IC design using SLOOP.

DANIEL A
The LC
is a tool
produces
based on
vides esse.
manually
topologic:
relationsh.
by selecti
use of the
gorithms (
addition t
During t
voltage gai
converted
meter val
circuit ele:
ments ba:
ing and si
mal layo:
schematic.
tween tran
the circuit
ulted. Ad.
entionally
location.
ities or
to rec
ements. (
forms con
ing ope
SCOAP [
gram) is
basic c
formed
logic i
ase or
input
value
and o
ling up
gates.
ability or
circuit
numbers.
proportiona
controllabili
test po
gates (
lity. At
the IBM LS:
designer
ques equi
gn based
The testal
onal and se
elate to the
ements w

The LOGMASC (LOGical MASk Checking) [21] program is a tool for design rule checking of IC masks. This program produces Boolean logic combinations of mask levels and is based on pattern recognition techniques. LOGMASC provides essentially all design rule checks which can be performed manually and provides pattern identification and extensive topological and geometrical information concerning the interrelationships of entities in the file. False errors are minimized by selectively applying design rule checks based on the circuit use of the particular patterns begin checked. Many of the algorithms can be used to solve other mask analysis problems in addition to the design rule checks.

During the circuit mask design process, transistor current or voltage gains, resistance, and capacitance values are scaled and converted to area definitions based on the nominal circuit parameter values used for circuit analysis. The actual sizes of the circuit elements and devices are adjusted to fit layout constraints based on minimum spacing, relative element partitioning, and size requirements. The actual circuit defined by the mask layout may be quite different than the original circuit schematic. For example, the interconnection distances between transistors introduce resistances and capacitances into the circuit which were not included in the initial circuit simulation. Additional parasitic transistors and diodes can be unintentionally introduced into the circuit which can cause improper operation. The CMAT [22] (Circuit MAsk Translator) code operates on circuit mask information, using pattern recognition to recognize and transcribe the mask areas into the circuit elements. CMAT employs the basic Boolean operations algorithms contained in LOGMASC and performs the required scaling operations to obtain circuit element values.

SCOAP [23] (Sandia Controllability Observability Analysis Program) is used to evaluate the testability of a circuit after the basic circuit design has been completed. This analysis is performed prior to layout and is based on a simple analysis of the logic interconnectivity. The SCOAP program calculates the ease or difficulty of setting an internal node from a primary input to a zero or one and the difficulty of observing a logic value at any internal node from a primary output. The zero and one values for controllability may be different depending upon the exact circuitry and interconnection of the logic gates. For nodes having very high values for controllability or observability changes are usually required in the logic circuitry to reduce the controllability and observability numbers. This reduction is required, since the numbers are proportional to the testing difficulty. For nodes with high controllability/observability measures, additional input or output te.: points or signal multiplexing may be required to set logic gates or observe node logic values to achieve circuit testability. At present, a structured design for testability, such as the IBM LSSD [24] approach, is not required. However, since the designer is responsible for designing a testable circuit, techniques equivalent to the LSSD approach may be used in the design based on the testability analysis information.

The testability analysis program is used for both combinational and sequential circuits. Combinational measures basically relate to the number of different line assignments or input assignments which must be made to set an internal node to a given value. For sequential circuits, the measures relate to the number of clock cycles or time frames required to propagate a specified logic value from an input pad to a node or from a specific node to an output terminal.

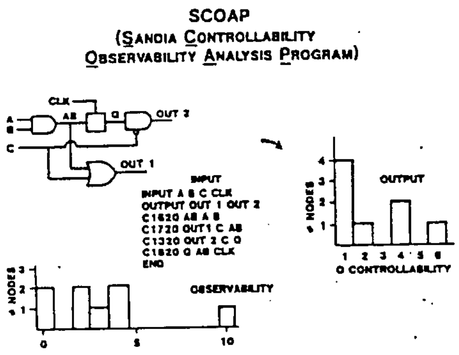


Fig. 12. Testability analysis using SCOAP.

An example at testability analysis is shown in Fig. 12. All input nodes are directly controllable as indicated for the four nodes with a "zero controllability" value of one. The most difficult node to control is usually an output node; for this example, output 2 has a zero controllability value of six. The observability is shown in the lower left of the figure. Again, the most observable nodes are the output nodes as indicated by the two nodes with an observability value of one. Ordinarily the most difficult nodes to observe are the input nodes and, therefore, the nodes $A$, $B$, $C$, and $CLK$ should be the least observable. In this case, the $CLK$ signal is input to a transmission gate between nodes $AB$ and $Q$; a change of state at node $AB$ must occur to observe the clock signal and, therefore, the $CLK$ signal has the large value of 10 for observability.

### *Data Base*

Because of the initial need for integrating loosely related CAD software and evolving design system specifications, a data-file base has been developed. The Sandia CAD data structure [25] is in the form of a "deciduous tree"; that is, one that can lose its leaves. This structure is a two-dimensional network with several restrictions. In one dimension the data structure forms a shallow tree one level deep. That is, any number of data types can be associated with a given node, but they may not be broken down further. In the other dimension, the structure is a network of nodes and subnodes to any depth. The leaves on the tree are the equivalent of small sequential files of data which can be retrieved by the user application program. This data file approach forces the designer to conform to a hierarchical structure while maintaining the freedom to store test files, documentation, etc., in the same file.

## SUMMARY AND FUTURE PLANS

Although a basic computer-aided LSI design capability has been established and is used by design engineers, the system is continuously evolving as new capabilities and enhancements are added. The modular programs and data structures, as well as the flexibility designed into the overall system framework, tend to minimize the cost for system modification as requirements change. In addition to continuous enhancement of existing programs, new aids for design synthesis, hierarchical simulation, symbolic layout, and test sequence generation will be developed. A continuing emphasis is placed on integrating the CAD software into a complete design system.

## References

[1] G. Kron, *Tensor Analysis of Network*. New York: Wiley, 1939.

[2] —, "A set of principles to interconnect the solutions of physical systems," *J. Appl. Phys.*, vol. 24, pp. 965–980.

[3] A. F. Malmberg, F. L. Cornwell, and F. N. Hofer, "NET-1 network analysis program," Los Alamos Rep. LA-3119, 1964.

[4] R. W. Jensen and M. D. Lieberman, *The IBM Circuit Analysis Program*. Englewood Cliffs, NJ: Prentice-Hall, 1968.

[5] A. Feller and M. D. Agostino, "Computer-aided mask artwork generation for IC arrays," in *Dig. 1968 IEEE Computer Group Conf.*, pp. 23–26.

[6] A. Feller, "Automatic layout of low-cost quick-turnaround random-logic custom LSI devices," in *Proc. 13th Design Automation Conf.*, pp. 79–85, June 1976.

[7] G. Persky, D. N. Deutsch, and D. G. Schweikert, "LTX—A system for the directed automatic design of LSI circuits," in *Proc. 13th Design Automation Conf.*, pp. 399–407, June 1976.

[8] E. Cohen, Program reference for SPICE 2, Memo ERL-M592, Electronics Research Laboratory, Univ. California at Berkeley, June 1976.

[9] M. R. Barbacci *et al.*, The ISPS computer description language, Tech. Rep Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, Mar. 1978.

[10] W. E. Cory *et al.*, An introduction to the DDL-P language, Tech. Rep. 163, Computer Systems Lab., Stanford Univ., Stanford, CA, Mar. 1979.

[11] W. M. van Cleemput, A structural design language for computer-aided design of digital systems, Tech. Rep. 136, Digital Systems Lab., Stanford Univ., Stanford, CA, Apr. 1977.

[12] J. Abraham, MINI—A logic minimization program, Univ. Illinois, Urbana, IL.

[13] DDA—Digital design aids for state machine synthesis, Tektronix, private communication.

[14] J. M. Acken and J. D. Stauffer, "Logic circuit simulation," *IEEE Circuits Syst. Mag.*, vol 1, no. 2, pp. 3–12, June 1979.

[15] S. P. Fan *et al.*, "MOTIS-C: A new circuit simulator for MOS LSI circuits," in *Proc. IEEE 1977 Int. Symp. on Circuits and Systems*, pp. 700–703, Apr. 1977.

[16] G. R. Boyle, *Simulation of Integrated Injection Logic*, Memo. UCB/ERL M78/13, Electron. Res. Lab., Univ. California at Berkeley, Mar. 1978.

[17] D. R. Alexander, R. J. Antinone, and G. W. Brown, *SPICE 2 Modeling Handbook*, BDM/A-77-071-TR, May 1977.

[18] B. T. Preas and W. M. van Cleemput, "Placement algorithms for arbitrarily shaped blocks," in *Proc. 16th Design Automation Conf.*, June 1979.

[19] B. T. Preas and W. M. vanCleemput, "Routing algorithms for hierarchical IC layout," in *Proc. Int. Symp. Circuits and Systems*, July 1979.

[20] SLOOP (Standard-Cell Layout Optimization Program), Sand Lab., to be published.

[21] B. W. Lindsay and B. T. Preas, "Design rule checking and analy of IC mask designs," in *Proc. 13th Design Automation Conf.*, June 1976.

[22] B. T. Preas *et al.*, "Automatic circuit analysis based on mask formation," in *Proc. 13th Design Automation Conf.*, June 197

[23] L. H. Goldstein, "Controllability observability analysis of digi circuits," *IEEE Trans. Circuits Syst.*, vol. CAS-26, Sept. 1979.

[24] E. B. Eichelberger and T. W. Williams, "A logic design structu for LSI testability," in *Proc. 14th Design Automation Conf.*, pp. 462–468, June 1977.

[25] J. D. Stauffer, "SADIST (The Sandia Data Index Structure)," Sandia National Lab., Albuquerque, NM, SAND80-1999, No 1980.

Marvin E. Daniel was born in Dexter, KS, o December 19, 1938. He received the B.S.E.E. degree from Kansas State University in 1961, the M.S.E.E. degree from University of New Mexico, Albuquerque, in 1963, and the Ph.D. degree in electrical engineering from Oklahom State University, in 1966.

In 1961, he joined Sandia National Laboratories, Albuquerque, where he has worked in a variety of positions including: design of te equipment, application of minicomputers automated test systems, development of mathematical models for CA programs, modeling of radiation effects in semiconductor devic analysis of weapon systems, economic studies of fusion and fossil fu energy systems, and for the past two-and-one-half years has been supe visor of the Computer-Aided Design Division in the IC Design a Fabrication organization. He is the author of numerous papers a articles, particularly in the areas of programs and models for comput aided design.

Charles W. Gwyn (M'68–SM'75) received th B.S. degree in electrical engineering from th University of Kansas, Lawrence, in 1961, a the M.S. and Ph.D. degrees in electrical eng neering from the University of New Mexic Albuquerque, in 1963 and 1968, respectively.

He joined Sandia National Laborator Albuquerque, in 1961, as a staff member whe he was engaged in many phases of nuclear radiation effects studies, including the study radiation effects on semiconductor devices analysis of the vulnerability of electrical systems to a radiation enviro ment. In 1973 he was assigned supervisory responsibility for th Computer-Aided Design and Analysis Division, and for develop computer techniques for use in the design, analysis, and layout the large scale integration (LSI) circuits. Currently he is departme manager for the IC Design Department and has responsibility for met oxide semiconductor (MOS) and bipolar IC design and testing, dev opment of computer aids, and development and procurement of spec semiconductor devices.

15

# Verifying Compiled Silicon

**Edmund K. Cheng**, Silicon Compilers, Inc., Los Gatos, CA

As the complexity of VLSI chips increases, the intuitions of design engineers fail more often. The large number of components and their complex relationships conspire to obscure the detailed consequences of each design decision. At the same time, the cost of making a mistake has escalated sharply, even in prototyping. To minimize the likelihood of costly errors, designers increasingly depend on computer tools to synthesize and verify designs before committing them to the costly fabrication process.

Verification tools fall into two categories: static and dynamic. Static tools check a design against a defined set of rules and flag any violations. Common static tools include design-rule checking (DRC), electrical-rule checking (ERC), and timing analysis. Dynamic tools (such as circuit, logic, and register-transfer-level simulators) imitate the behavior of a design, thus allowing the engineer to test and understand a circuit or system in a variety of environments. Conventional IC design requires three manual translations in traversing the four design steps from architecture to logic to transistor circuit to layout. As a result, most of the design verification efforts focus on establishing the equivalence of the different design representations.

Silicon Compilers, Inc. recently introduced a system (Genesil Silicon Development System) that shifts the process of designing VLSI chips towards the architectural level and away from the classical IC-design considerations of logic and layout topology. Such a shift implies changes in verification methods as well as design methods. This article describes VLSI design verification in the context of this silicon compiler, namely, functional simulation and timing analysis. We treat the two functions separately, describe how they work, and contrast them with conventional verification.

## Silicon Compilation

Just as software compilation synthesizes machine-language code from programs written in a high-level language, the Genesil Silicon Development System synthesizes relevant *views* of an IC from high-level architectural descriptions. These views are derived from descriptions in terms of structures "known" to the compiler, such as PLAs, ROMs, RAMs, ALUs, registers, and data paths. These structures can be described functionally, in quantity, or in terms of mathematical, logical, or state equations. The design system then synthesizes views of the layout (IC topology), the function (simulation model), and the performance (timing model) of the specified architecture.

The physical circuitry is algorithmically generated using layout/circuit primitives that have been verified previously (using conventional verification tools). Layout design of the "known" structures in this system can therefore be characterized as "correct-by-construction." Architectures may be specified hierarchically and the layouts may be constructed hierarchically (wiring blocks into modules and modules into chips). Assembling modules of compiled blocks requires manual placement of block outlines. The system then automatically routes interconnections.

| Equivalence Check | Traditional Method | Silicon Compilation |
|---|---|---|
| Wish to Architecture | RTL Simulation | Functional Simulation |
| Architecture to Logic | Logic Simulation | (implicit) |
| Logic to Circuit | Switch Simulation | (implicit) |
| Circuit to Layout | Extraction | (implicit) |
| Layout (Absolute) | DRC | (implicit) |
| Layout to Wish | Circuit Simulation | Timing Analysis |

**TABLE 1. Levels of design and verification.**

At the architectural level, functionality is verified through functional simulation, while the circuit performance is verified through timing analysis. The functional and circuit models that are used for design verification are automatically synthesized from the design specifications. Typically, the major portion of the development time is spent on these two verification activities. Because the system synthesizes the simulation and timing models automatically as part of module compilation, the design engineer may move with relative ease between design specification and design verification. This ease encourages a design method that is a process of successive, or incremental, refinement.

## Separation of Functional Simulation and Timing Analysis

Table 1 shows the conventional levels of IC design and their respective verification operations. As mentioned above, most of the verification efforts in conventional IC design are concentrated on checking the equivalence between the various design levels. Silicon compilation, on the other hand, replaces all of the manual steps between architecture design and layout design, thus obviating the associated equivalence checking. Instead, this design process requires only functional simulation at the architectural level (to verify the architectural design) and timing analysis at the layout level (to verify the timing performance).

Designers commonly handle functional and timing designs separately, even though the two are to some extent intermingled in an iterative process. When working with synchronous circuits, it is often possible to treat function and timing as separable processes, and thereby deal with only one variable at a time. We strictly separate functional simulation and timing analysis functions in the Genesil system, because the two differ markedly in their goals and methods.

Because a large number of test vectors typically are required to adequately verify all the functions of a large digital network, simulation can be rather time-consuming. Therefore, it is extremely important to minimize the compute time required for each cycle of simulation. For this reason, simulation should be done at the highest possible level (without sacrificing accuracy). Computing timing delays concurrently with the evaluation of functional behavior not only may not yield absolutely accurate timing results due to circuit interactions, but also may further burden the computational requirements for each simulation cycle. While the test vectors are driving the functional simulation to exercise all the functions of the circuitry, timing delay paths could be computed redundantly if they were tagged along in the calculations. On the other hand, a dedicated timing analysis program that evaluates each timing delay path only once would be much more efficient in terms of compute time.

While the compute-time resource is a problem when many test vectors must be used to verify function, the engineering resource in generating those vectors is an even bigger problem. When a design engineer is running a functional or logic simulator, he focuses on searching out the problems in the function of the design. Since his main concern is to design the test vectors for that purpose only, the test vectors may not exercise all of the critical delay paths in the circuit. Hence, some critical short paths (race conditions) and long paths (delay times) may go unnoticed.

On the other hand, a timing analysis program can exhaustively analyze all possible timing paths in the circuit. With pruning of irrelevant paths by the designer, this approach has been found to be feasible. In contrast, it is not feasible to generate automatically test vectors for functional simulation, because it is not possible for a machine to know the intent of the design.

Another distinction argues for the separation of functional and timing analyses: Whereas functional simulation can usually be performed without reference to the process technology, timing analysis calculations depend in detail on physical parameters.

## Functional Simulation

For a large digital network, a number of test vectors on the order of 100,000 is not uncommon. If the circuit is modeled at the gate or switch level, the CPU run time and memory required for each cycle of simulation are very high. Table 2 quantifies this statement for one instruction cycle in a VLSI chip (Kleckner *et al.* 1982).

Silicon compilation focuses the process of designing VLSI chips towards the architectural level, which can be conveniently simulated with functional models. Typically, the majority of a chip design would be composed of complex functional blocks, while the "odds and ends" would be implemented using gate-level blocks. The blocks that are

| Level of Simulation | CPU Time | Memory (MB) |
|---|---|---|
| RTL | <<1 min | <<0.5 |
| Logic | 10 min | 4 |
| Timing | 8 hours | 30 |
| Circuit | 6 months | 250 |

**TABLE 2. Comparison of run time and memory requirements for simulation of one instruction cycle in a VLSI chip.**

defined at the functional level do not have to be simulated at the gate or switch level, because functionality at these levels is guaranteed by the compilation process. Performance verification, one of the classical imperatives for low-level simulation, is accomplished in an entirely separate step, via timing analysis, as discussed below.

Genesil's functional simulator is implemented using the selective trace event-driven technique, which permits multiple system clocks. Signal values are evaluated between clock edges; in other words, the clock cycles are assumed to be long enough so that all signals stabilize before the clock edges. (The minimum clock cycle times are determined in timing analysis.) Signal values are evaluated by calling block models that dynamically interpret their respective functional parameter specifications.

The simulator's user interface enables the designer to observe and/or modify the state of the simulation network and environment by defining or deleting clock cycle references; setting up and invoking checkpoints; defining vectors and mnemonics; issuing general resets; loading or clearing the contents of a RAM, ROM, or PLA array; and setting up test vectors. In addition, an interactive screen interface can trace signal histories and allow the user to observe and alter current network states. Various screen formats may be defined by the designer for use in interactive simulations or in viewing results of batch-mode runs.

## Timing Analysis

Timing performance in conventional VLSI design commonly has been determined by extracting all the parametric details of the layout of a circuit and then feeding those parameters to a circuit simulation model of the portion of the circuit under study. Table 2 indicates the exorbitant amount of compute time needed to simulate VLSI chips at the circuit level. The magnitude of this expense makes it impractical to use circuit-level simulation to obtain timing data, except for small and isolated pieces of circuitry.

Therefore, the design engineer usually identifies manually portions of the circuit whose delay times he deems to be the worst cases. Due to the complexity of VLSI designs, such manual efforts tend to miss some critical paths, either because of oversight in path selection or mistakes in the circuit design. Furthermore, circuit simulators are data-dependent; the timing information produced depends on the input stimuli. To eliminate these problems, analysis must be performed on the entire chip in a data-independent fashion. However, in order for such automatic path enumeration to be practical, the timing analysis must run much faster than circuit simulation does.

Several timing analysis algorithms have been reported recently (Jouppi 1983; Ousterhout 1983; Lin *et al.* 1984). In terms of

CPU time, a timing analysis program runs up to 10,000 times faster compared to a circuit simulation program such as SPICE, while yielding results that are very close to those that a circuit simulator would estimate for most delay paths.

Because of their speed, timing analyzers can be used for checking all possible timing paths in a VLSI chip. Unlike a circuit simulator, which is driven by external stimuli provided by the design engineer, a timing analyzer automatically enumerates all possible timing paths, analyzes their timing delays, and reports on the worst of them. Hence, this approach is not vulnerable to the design engineer's fallibility in picking out critical paths.

Although timing analysis is not as accurate as circuit simulation, through careful crafting of the device models, adequate results can be obtained. For example, the Genesil Timing Analyzer is fast (2000 transistors per minute) and accurate (± 10% of SPICE results). The program requires no test vectors, and is exhaustive in its path analysis.

However, in general, not all of the topologically possible delay paths represent signal paths that can occur in practice. Therefore, a timing analysis system must provide a user complete control to filter out unwanted or illogical results. In setting up a timing analysis in Genesil, the user may bind nodes to fixed values, force the direction of signal flow on bidirectional wires, flag nodes to be ignored, and hard-wire delays between two nodes.

To obtain accurate timing results, routing should be completed on the module or chip, because interonnection impedances significantly affect timing. However, the timing of portions of a module may be analyzed as it is being composed by estimating the external load impedances. To do so, a routing estimation is performed or else estimated models of the external circuitry are provided. Such estimations can later be confirmed by analyzing the finished module.

When timing analysis of a synthesized block is requested, the analyzer examines every possible path through the circuitry (subject to the user constraints mentioned above) to determine the worst-case delay times. The system uses built-in knowledge about which parameters are most important to produce a timing data-sheet, similar to what one would see in a typical component "data book". Several reports are provided:

1. A *timing* report identifies those paths that limit the minimum clock periods and duty cycles (rank-ordered from worst to best), lists the corresponding clock period and duty cycle values and the total cycle time, displays the input set-up and hold times, and lists the output delay times with respect to a clock transition.
2. A *node-delay* report lists the maximum and minimum delay times for a given node with respect to the clocks.
3. A *delay-path* report lists the paths that set the maximum and minimum delay between any two nodes.

If, after reviewing the timing report, the engineer finds unsatisfactory delays in portions of the circuit, he can attempt to refine the design. He may rearrange the block placements in order to shorten wiring lengths of critical signals. Or, he may redesign the circuit. For example, long delay paths can be broken by latches, creating a pipeline.

The Genesil Timing Analyzer works from a timing model that is synthesized by the system at the time the block or module is compiled. The model consists of a representation of the circuit at the transistor level, including all intrinsic resistances and capacitances, and those parasitic values contributed by the interconnection network.

As a result, the attributes of transistors and nodes consist not only of information about themselves, but also of information about their environment and how they are used. For example, the model resolves the directions of all signal flows, and classifies all transistors into one of several types, such as precharging transistors, bootstrap drivers, discharging transistors, or pull-ups. Clearly, the customized device models are crucial to the accuracy of the timing analyzer and also depend in detail on the underlying technology. In order to ease the job of calibrating these models, they are stored in look-up tables (called a "fabline file") instead of being coded into the program algorithm. Thus, the timing analyzer is accurate in terms of the fabrication line specified for the block, module, or chip. By extension, if the system is calibrated for several lines, a designer can compare the predicted performance of a circuit from several prospective foundries by changing the fab-line parameter and recompiling.

## Conclusions

Logic design is a process of incremental refinement, in which design specifications and design goals co-evolve. The design method embodied in the Genesil system requires the user to perform a functional simulation at the architectural level and subsequently, a static timing analysis based upon a model of the final layout. The functional simulation is efficient in terms of computer resources and "easier" than a logic simulation, because the number of structures being simulated in a typical VLSI chip will be less than 100, rather than in the tens of thousands. The static timing analysis runs much faster than a circuit simulation does and automatically enumerates all possible delay paths. Performance verification reduces to a process of checking whether a worst-case path satisfies the design specification. ☐

## References

Jouppi, N. 1983. "TV: An nMOS Timing Analyzer." *Third Caltech Conference on Very Large-Scale Integration*. Computer Science Press. Rockville, MD.

Kleckner, J.E., R.A. Saleh, and A.R. Newton. 1982. "Electrical Consistency in Schematic Simulation." *ICCC*.

Ousterhout, J. 1983. "Crystal: A Timing Analyzer for nMOS VLSI." *Third Caltech Conference on Very Large-Scale Integration*. Computer Science Press. Rockville, MD.

Lin, T. and C.A. Mead. October 1984. "Signal Delay in General RC Networks." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

## About the Author

**Edmund K. Cheng** is the Vice President of Engineering at Silicon Compilers, Inc. He received the BSEE degree from Ohio University and the MSEE and Ph.D. from the California Institute of Technology. For six years he worked in the microcomputer design engineering department at Intel, where he developed the A/D converters for single-chip microcomputers and managed automotive-engine-control projects. Since 1981, Ed has led the development of the Genesil silicon compiler.

DEF011950

16

IBM

# Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

DEF083932

7 8 9 0 DOC/DOC 9 9 8

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

*The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.*

*Printed and bound by R. R. Donnelley & Sons Company.*

**Tenth Edition (August 1993)**

This is a major revision of the *IBM Dictionary of Computing,* SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

This book is printed on acid-free paper.

**compound string** A type of string designed to simplify national language support by allowing text to be displayed without hard-coding language-dependent attributes such as character sets and text.

**compress** (1) In a character string, to reduce the space taken on a data medium by repetitive characters. (T) (2) To save storage space by eliminating gaps, empty fields, redundancy, or unnecessary data to shorten the length of records or files. (3) To move files, libraries, or folders together on disk to create a continuous area of unused space.

**compressed audio** Audio resulting from the process of digitally encoding and decoding up to 40 seconds of voice-quality audio for each individual videodisc, resulting in a potential for over 150 hours of audio per 12-inch videodisc. Synonymous with still-frame audio.

**compressed disk file** In System/36, a file that contains unprocessed records.

**compressed encoding** (1) A process in which a contiguous string of bits, characters, or data units is reduced to a shorter string in such a way that a second contiguous string, yielding the same short string, cannot be found and the process cannot be reversed. (2) A process in which variable-length messages are reduced to a shorter, fixed-length message.

**compressed pattern storage** Storage that holds the extended fonts for the 3800 printer.

**compressed video** Video resulting from the process of digitally encoding and decoding a video image or segment using a variety of computer techniques to reduce the amount of data required to represent the content accurately.

**compression** (1) The process of eliminating gaps, empty fields, redundancies, and unnecessary data to shorten the length of records or blocks. (2) In SNA, the replacement of a string of up to 64 repeated characters by an encoded control byte to reduce the length of the data stream sent to the LU-LU session partner. The encoded control byte is followed by the character that was repeated (unless that character is the prime compression character). See also compaction, string control byte. (3) In Data Facility Hierarchical Storage Manager, the process of moving data instead of allocated space during migration and recall in order to release unused space. (4) Contrast with decompression.

**compressor** An electronic device that compresses the volume range of a signal. See also compandor, expandor.

**COM printer** A page printer that produces on a photographic film a microimage of each page. Synonymous with computer output microfilm printer. (T) (A)

**compromise** In computer security, a violation of the security policy of a system in which unauthorized intentional or unintentional disclosure, modification, or destruction, or loss, of an object, may have occurred.

**compromise net** A network, used in conjunction with a hybrid coil to balance a subscriber's loop, that is adjusted for an average loop length or an average subscriber's set, or both, to secure compromise (not precision) isolation between the two directional paths of the hybrid.

**compromising emanations** In computer security, unintentional intelligence-bearing signals that may convey data and that, if intercepted and analyzed, may compromise sensitive information being processed or transmitted by a computer system.

**COMPUSEC** Computer security.

**computational stability** The degree to which a computational process remains valid when subjected to effects such as errors, mistakes, or malfunctions. (A)

**compute mode** That operating mode of an analog computer during which the solution is in progress. Synonymous with operate mode. (T)

**computer** A functional unit that can perform substantial computations, including numerous arithmetic operations and logic operations without human intervention during a run. In information processing, the term computer usually describes a digital computer. A computer may consist of a stand-alone unit or may consist of several interconnected units. (T)

**computer abuse** In computer security, a willful or negligent unauthorized activity that affects the availability, confidentiality, or integrity of computer resources. Computer abuse includes fraud, embezzlement, theft, malicious damage, unauthorized use, denial of service, and misappropriation. See also information system abuse.

**computer-aided (CA)** Pertaining to a technique or process in which part of the work is done with the assistance of a data processing system. Synonymous with computer-assisted. (T)

**computer-aided design (CAD)** The use of a computer to design or change a product, tool, or machine, such as using a computer for drafting or illustrating.

*Note:* Sometimes, CAD and CAM are used together and expressed as CAD/CAM. (T)

**computer-aided engineering (CAE)** Analysis of a design to check for basic errors, or to optimize manufacturability, performance or economy; for example, by comparing various possible materials or designs.

*Note:* Information from the CAD/CAM design database is used to analyze the functional characteristics of a part, product under design, and to simulate its performance under various conditions. (T)

**computer-aided industry (CAI)** The use of computer systems to assist in the operation of an industry. (T)

**computer-aided instruction (CAI)** The use of a computer to assist human instruction. (T) Synonymous with computer-assisted instruction.

**computer-aided manufacturing (CAM)** The use of computer technology to direct and control the manufacturing process. (T)

**computer-aided planning (CAP)** All activities for preparation of the basic data about production processes by usage of computer technology. (T)

**computer-aided publishing** Synonym for electronic publishing. (T)

**computer-aided quality assurance (CAQ)** The use of computer technology to plan, monitor and control processes, parts and products throughout all phases of the product life cycle; this includes an overall quality report system from design to field performance and from shop floor to the management. (T)

**computer-aided retrieval (CAR)** Systems that combine the document storage capabilities of micrographics with the indexing and retrieval capabilities of a computer database.

**computer-aided software engineering (CASE)** (1) The automation of well-defined methodologies that are used in the development and maintenance of products. These methodologies apply to nearly every process or activity of a product development cycle, examples of which include project planning and tracking, product designing, coding, and testing. (2) A set of computer-based development tools to automate certain portions of methodologies. Thus, CASE tools work within a methodology rather than compose a methodology themselves. See also CCASE, ICASE.

**computer-animated graphics** In multimedia applications, graphics animated by means of a computer, rather than videotape or film.

**computer architecture** (1) The logical structure and functional characteristics of a computer, including the interrelationships among its hardware and software components. (T) (2) The organizational structure of a computer system, including hardware and software. (A) (3) See also hypercube, parallel processor architecture.

**computer-assisted** Synonym for computer-aided (CA). (T)

**computer-assisted instruction (CAI)** Synonym for computer-aided instruction.

**computer-assisted publishing** Synonym for electronic publishing. (T)

**computer-based training** Synonym for computer-assisted instruction.

**computer center** A facility that includes people, hardware, and software, organized to provide information processing services. Synonymous with data processing center, installation. (T) (A)

**computer conferencing** Computerized communication that allows people distant from each another to enter and receive text and graphic messages via interconnected terminals. (T) See also conference call, teleconferencing, videoconferencing.

**computer crime** (1) In computer security, a crime committed through the use of software or data. (T) (2) A crime committed through the use of software or data residing in a computer. (T) (A)

**computer cryptography** In computer security, the use of a cryptographic algorithm in a computer to perform encryption and decryption to protect information or to authenticate users, sources, or information.

**computer edit system** A video editing system, controlled by a computer and connected to machines for recording and playback.

**computer-dependent language** Synonym for computer-oriented language.

**computer fraud** (1) In computer security, a computer crime that involves deliberate misrepresentation or alteration of data in order to obtain something of value, usually for monetary gain. (T) (2) Deception by means of a computer, deliberately practiced in order to secure unfair or unlawful gain. (A)

**computer generation** A category in a historical classification of computers based mainly on the technology used in their manufacture; for example, first generation based on relays or vacuum tubes, the second on transistors, the third on integrated circuits. (T)

**operating time** (1) That part of operable time during which a functional unit is operated. (A) (2) Contrast with idle time.

**operating voltage indicator** On a calculator, a device giving a visual signal to indicate that the correct voltage is set for a main-powered machine or that the battery is insufficiently charged in a battery-powered machine. (T)

**operation** (1) A well-defined action that, when applied to any permissible combination of known entities, produces a new entity; for example, the process of addition in arithmetic; in adding five and three and obtaining eight, the numbers five and three are the operands, the number eight is the result, and the plus sign is the operator indicating that the operation performed is addition. (I) (A) (2) A defined action, namely, the act of obtaining a result from one or more operands in accordance with a rule that completely specifies the result for any permissible combination of operands. (A) (3) A program step undertaken or executed by a computer; for example, addition, multiplication, extraction, comparison, shift, transfer. The operation is usually specified by the operator part of an instruction. (A) (4) The event or specific action performed by a logic element. (A) (5) An action performed on one or more data items, such as adding, multiplying, comparing, or moving. (6) In object-oriented design or programming, a service that can be requested at the boundary of an object. Operations include modifying an object or disclosing information about an object.

**operational amplifier** A high-gain amplifier connected to external elements to perform specific operations or functions. (I) (A)

**Operational Assistant** In the AS/400 system, a part of the operating system that provides a set of menus and displays for end users to do commonly performed tasks, such as working with printer output, messages, and batch jobs.

**operational diskette** Synonym for working diskette.

**operational environment** (1) The physical environment; for example, temperature, humidity, and layout. (2) All of the IBM-supplied basic functions and the user programs that can be executed by a store controller to enable the devices in the system to perform specific operations. (3) The collection of IBM-supplied store controller data, user programs, lists, tables, control blocks, and files that reside in a subsystem store controller and control its operation. (4) See also configuration image.

**operational expression** In PL/I, an expression that consists of one or more operations.

**operational key** Synonym for session cryptography key.

**operational mode** See asynchronous balanced mode, asynchronous response mode, normal response mode.

**operational rights** The authority to use an object and to look at its description.

**operational sign** An algebraic sign associated with a numeric data item or a numeric literal that indicates whether the item is positive or negative.

**operational unit (OU) number** In System/36, the number that corresponds to the line connector, located on the back of the system unit, to which a line is attached.

**operation code** (1) A code for representing the operation parts of the machine instructions of a computer. (T) (2) A code used to represent the operations of a computer. (3) In SSP-ICF, a code used by a System/36 application program to request SSP-ICF data management or the subsystem to perform an action; for example, the operation $$SEND asks that data be sent. (4) In RPG, a word or abbreviation, specified in the calculation specifications, that identifies an operation.

**operation code trap** A specific value that replaces the normal operation part of a machine instruction at a particular location to cause an interrupt when that machine instruction is executed. (T)

**operation control language (OCL)** A programming language used to code operation control statements.

**operation control statement** A statement in a job or job step that is used in identifying the job or describing its requirements to the operating system.

**operation decoder** A device that selects one or more control channels according to the operation part of a machine instruction. (A)

**operation expression** An expression containing one or more operators.

**operation mode** The normal working state of a product or system. See also maintenance mode.

**operation part** (1) The part of a machine instruction or microinstruction that specifies the operation to be performed. (T) (2) The part of an instruction that specifies the operation to be performed. Synonymous with function part, operator part. (A) (3) See also implied addressing.

**operations analysis** Synonym for operations research.

**RTAM generation** The process of assembling selected RTAM facilities and link editing them into VS1.

**RTB** Response/throughput bias.

**RTG** Route Table Generator.

**RTM** Realtime monitor.

**RTTY** Radio teletypewriter telecommunications.

**RTV** Real-Time Video.

**RU** Request/response unit.

**rubber-banding** In computer graphics, moving the common ends of a set of straight lines while the other ends remain fixed. (I) (A) See Figure 129.

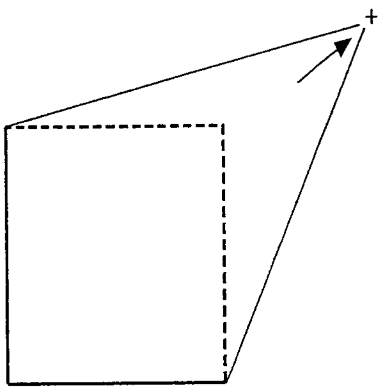


Figure 129. Rubber-Banding

**rubber-band outline** In the AIX operating system, a window with a moveable outline.

**rub-out character** Synonym for delete character.

**RU chain** In SNA, a set of related request/response units (RUs) consecutively transmitted on a particular normal or expedited data flow. The request RU chain is the unit of recovery. If one RU in the chain cannot be processed, the entire chain must be discarded.

*Note:* Each request unit belongs to only one chain, which has a beginning and an end indicated through control bits in request/response headers within the RU chain. Each RU can be designated as first-in-chain (FIC), last-in-chain (LIC), middle-in-chain (MIC), or only-in-chain (OIC). Response units and expedited-flow request units are always sent as only-in-chain.

**rule** A solid or patterned line of any weight, extending horizontally or vertically across a column or row.

**rule-based system** A computer system which performs inferences by applying a set of if then rules to a set of facts following given procedures. Synonymous with production system. (T)

**rule interpreter** Synonym for inference engine. (T)

**ruler line** A line that indicates where the left and right margins and any tab stops are set. (T)

**run** (1) A performance of one or more jobs. (I) (A) (2) A performance of one or more programs. (I) (A) (3) To cause a program, utility, or other machine function to be performed.

**runaway task** In CICS, a task that does not relinquish control within an interval of time defined by the user.

**RUN disk** The virtual disk that contains the VTAM, NetView, and VM/SNA console support (VSCS) load libraries, program temporary fixes (PTFs) and user-written modifications from the ZAP disk. See BASE disk, DELTA disk, MERGE disk, ZAP disk.

**rundown** In multimedia, an outline of the content of a video program for which a script is inappropriate or impossible, such as an interview.

**run duration** Synonym for running time.

**run file** In the AIX operating system, synonym for load module.

**run-length coding** A technique for compressing data that avoids having to code repeatedly data elements of the same value; instead, the value is coded once, along with the number of times for it to be repeated.

**run list** In VM/SP, a list of virtual machines that are receiving a queue slice on the processing unit. The virtual machine currently executing is called the runuser. When virtual machines are dropped from the run list, replacement is made from the eligible list. See also dispatch list, eligible list.

**running foot** (1) Synonym for footer. (T) (2) A footing that is repeated above the bottom margin area on consecutive pages or on consecutive odd-numbered or even-numbered pages in the text area of the page. Synonymous with footer.

**running heading** (1) Synonym for header. (T) (2) A heading that is repeated below the top margin area on consecutive pages or on consecutive odd-numbered or even-numbered pages in the text area of the page.

**running open** In telegraph applications, a term used to describe a machine connected to an open line or a line without battery (constant space condition). A telegraph receiver under such a condition appears to